Apple III

SOS

Reference Manual, Volume 2

apple computer
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-0442-B

Customer Satisfaction

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the documentation or media at no charge to you during the 90-day period after you purchased the product.

In addition, if Apple releases a corrective update to a software product during the 90-day period after you purchased the software, Apple will replace the applicable diskettes and documentation with the revised version at no charge to you during the six months after the date of purchase.

In some countries the replacement period may be different; check with your authorized Apple dealer. Return any item to be replaced with proof of purchase to Apple or an authorized Apple dealer.

Limitation on Warranties and Liability

Even though Apple has tested the software described in this manual and reviewed its contents, neither Apple nor its software suppliers make any warranty or representation, either express or implied, with respect to this manual or to the software described in this manual, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this software and manual are sold "as is", and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its software suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software or manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Copyright

Product Revisions

Apple cannot guarantee that you will receive notice of a revision to the software described in this manual, even if you have returned a registration card received with the product. You should periodically check with your authorized Apple Dealer.

20525 Mariani Avenue
Cupertino, California 95014

Apple and the Apple logo are registered trademarks of Apple Computer, Inc.

Simultaneously published in the U.S.A and Canada.

Reorder Apple Product #A3L0027-A

Apple III

SOS Reference Manual

Volume 2: The SOS Calls

Acknowledgements

Knuth, *Fundamental Algorithms: The Art of Computer Programming,* Vol. I, 2/e.© 1981. Reproduction of book cover. Reprinted with permission.

Writer: Don Reed

Contributions and assistance: Bob Etheredge, Tom Root, Bob Martin, Dick Huston, Steve Smith, Dirk van Nouhuys, Ralph Bean, Jeff Aronoff, Bryan Stearns, Russ Daniels, Lynn Marsh, and Dorothy Pearson

Contents

Volume 2: The SOS Calls

Figures and Tables

Volume 2: The SOS Calls

Preface

Volume 2: The SOS Calls comprises the remaining chapters and the appendixes of this manual. The chapter numbers continue the sequence of those in Volume 1.

Volume 2 defines the individual SOS calls. Chapter 9 contains a description of each file call; Chapter 10, each device call; Chapter 11, each memory call; and Chapter 12, each utility call. Each of these chapters is divided into two sections: calls, and errors.

The calls defined in each chapter are arranged in numerical order by call number (for example, CREATE is $CØ). Each call description contains the following information:

- Definition of the call
- Required parameters
- Optional parameters
- Comments
- Errors

The parameter fields are of four types:

- Pointer (2 bytes): The location of a table or parameter list.
- Value (1, 2, or 4 bytes): A parameter passed by the caller to SOS.
- Result (1, 2, or 4 bytes): A parameter returned by SOS to the caller.
- Value/result (1, 2, or 4 bytes): A parameter passed to SOS and back to the caller, possibly changed.
- Unused (any length): Occurs when the same parameter list is used by two calls, one of which ignores some parameters in the list. An unused field can be of any length.

Each SOS call has three parts, described in Chapter 8 of Volume 1:

- The call block
- The required parameter list
- The optional parameter list

They can be diagrammed as shown in Figure 0-1:

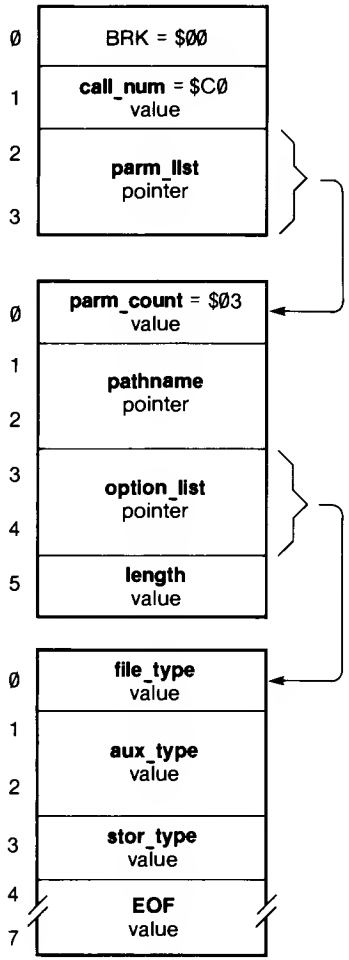


Figure 0-1. Parts of the SOS Call

Each call description is accompanied by a diagram like that shown in Figure 0-1. Most of the diagrams omit the call block, as these are identical, except for the **call_num** , and show only the required and optional parameter lists. In addition, the **parm_count** (shown in the diagram) is omitted from the required parameter list.

The one exception to this pattern is TERMINATE, for which the call block only is shown, as in Figure 0-2, because it differs from the standard form. See section 12.1.6 for details.

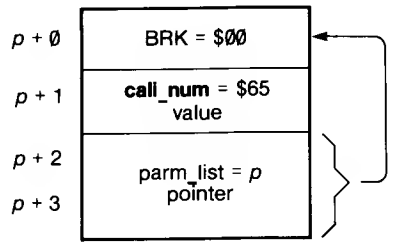


Figure 0-2. TERMINATE Call Block

9

File Calls and Errors

9.1 File Calls

This section contains descriptions of all calls that operate on files. These calls operate on closed files and refer to a file by its pathname.

$C0: CREATE
$C1: DESTROY
$C2: RENAME
$C3: SET_FILE_INFO
$C4: GET_FILE_INFO
$C5: VOLUME
$C6: SET_PREFIX
$C7: GET_PREFIX
$C8: OPEN

These calls operate on access paths to open files and refer to the access path by its **ref_num**, returned by the OPEN call.

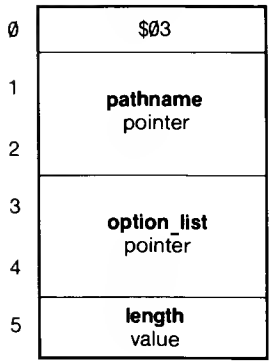
$C9: NEWLINE
$CA: READ
$CB: WRITE
$CC: CLOSE
$CD: FLUSH
$CE: SET_MARK
$CF: GET_MARK
$D0: SET_EOF
$D1: GET_EOF
$D2: SET_LEVEL
$D3: GET_LEVEL

9.1.1 CREATE — File Call $C0

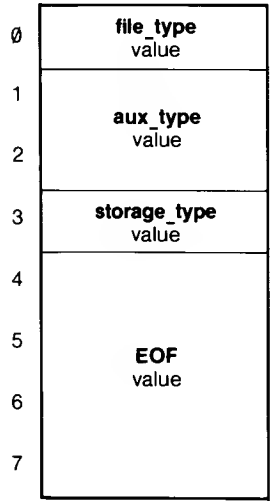
This call creates a standard file or subdirectory file on a volume mounted on a block device. A directory entry is established, and at least one block is allocated on the volume.

This call cannot create a volume directory or a character file. Volume directories are "created" by the formatting utility on the Apple III Utilities disk. Character files are "created" by the System Configuration Program.

CREATE $C0



Required Parameter List



pathname: pointer

This parameter is a pointer to a string in memory containing the pathname of the file to be created: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The last name in the pathname should be that of a file that does not currently exist in the specified directory, or a DUPERR will result.

option_list: pointer

This is a pointer to the optional parameter list if **length** (below) is between 1 and 8; otherwise it is ignored.

length: 1 byte value
Range: $0..$08

This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.

The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.

Ø = no optional parameters
1 = **file_type**
3 = **file_type** through **aux_type**
4 = **file_type** through **stor_type**
8 = **file_type** through **EOF**

Optional Parameter List

file_type: 1 byte value
Range: $ØØ..$FF
Default: $ØØ

This is the type identifier for this file. The **file_type** does not affect the way in which SOS deals with the file: it is used only by interpreters to determine the internal arrangement and meaning of the bytes in the file. These values of **file_type** are now defined:

$ØØ = Typeless file (BASIC or Pascal "unknown" file)
$Ø1 = File containing all bad blocks on the volume
$Ø2 = Pascal or assembly-language code file
$Ø3 = Pascal text file
$Ø4 = BASIC text file; Pascal ASCII file
$Ø5 = Pascal data file
$Ø6 = General binary file
$Ø7 = Font file
$Ø8 = Screen image file
$Ø9 = Business BASIC program file
$ØA = Business BASIC data file
$ØB = Word Processor file
$ØC = SOS system file (DRIVER, INTERP, KERNEL)
$ØD, $ØE = SOS reserved
$ØF = Directory file (see **storage_type**)
$1Ø..$DF = SOS reserved
$EØ..$FF = ProDOS reserved

aux_type: 2 byte value
Range: $ØØ..$FFFF
Default: $ØØØØ

This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (**storage_type** is $ØF), these bytes contain the total number of blocks on the volume.

storage_type: 1 byte value
Range: $Ø1..$ØD
Default: $Ø1

This indicates whether the file is to be a standard file ($Ø1) or a subdirectory file ($ØD). All other values are illegal and will result in a TYPERR.

EOF: 4 byte value
Range: $ØØØØØØØØ..$ØØFFFFFF
Default: $ØØØØØØØØ

This specifies the amount of space to preallocate for the file.One data block is automatically allocated regardless of the value of **EOF**; additional data blocks are allocated until the number of bytes in the allocated data blocks equals or exceeds **EOF**. In addition to the data blocks, index blocks are allocated as necessary.

The maximum creation size for standard files is $ØØFFFFFF, or $8ØØØ blocks. The maximum creation size for subdirectories is $ØØØØFFFF, or $8Ø blocks. The total number of blocks occupied by a file is the number of data blocks plus the number of index blocks: see Chapter 5 of Volume 1 for more information.

Comments

The file created must be a block file. The **access** attribute of the file is implicitly set to the following:

standard file = $E3: (destroy, backup, rename, write, read)
subdirectory = $E1: (destroy, backup, rename, NO write, read)

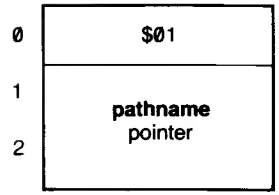
Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$40:	BADPATH	Invalid pathname syntax
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	Subdirectory file not found
$47:	DUPERR	Attempt to CREATE an existing file
$48:	OVRERR	Overrun error. Either EOF too large or not enough disk space
$49:	DIRFULL	Directory is full
$4B:	TYPERR	**Storage_type** parameter neither $01 nor $0D
$52:	NOTSOS	Not a SOS volume
$53:	BADLSTCNT	Invalid **length** parameter
$58:	NOTBLKDEV	Not a block device

9.1.2 DESTROY File Call $C1

This call deletes the file specified by the **pathname** parameter by removing the file's directory entry. DESTROY releases all blocks used by that file back to free space on that volume.

DESTROY $C1



The file can be either a standard or subdirectory file. Volume directories cannot be destroyed except by physical reformatting of the medium. Character files are "destroyed" by the System Configuration Program.

Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the pathname of the file to be destroyed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

Comments

A file cannot be destroyed if it is currently open. If the **pathname** refers to a subdirectory file, then that subdirectory must be completely empty in order for the subdirectory to be destroyed.

Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$40:	BADPATH	Invalid pathname syntax
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	File not found
$4A:	CPTERR	Incompatible file format
$4B:	TYPERR	Unsupported file storage type
$4E:	ACCSERR	File's **access** attribute prevents DESTROY
$50:	FILBUSY	File is open. Request denied.
$52:	NOTSOS	Not a SOS volume
$58:	NOTBLKDEV	Not a block device

9.1.3 RENAME File Call $C2

This call changes the name of the file specified by the **pathname** parameter to that specified by **new_pathname**. Only block files may be renamed; character files are "renamed" by the System Configuration Program.

RENAME $C2

0	$02
1	**pathname** pointer
2	
3	**new_pathname** pointer
4	

Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the old pathname of the file to be renamed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The **pathname** must refer to either a volume directory, subdirectory, or standard file.

new_pathname: pointer

This parameter is a pointer to a string containing the new pathname of the file to be renamed: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. The pathname can be either a complete or partial pathname. Only the last file name of the new pathname may differ from that in the old pathname.

Comments

The file must reside on a block device. Both **pathname** and **new_pathname** must be identical except for the last file name. For example, the path /VOL.1/FILE.1 can be renamed /VOL.1/FILE.2 , but not /VOL.2/FILE.X or /VOL.1/SUBDIR.A/FILE.X .

A file may not be renamed while it is open for writing.

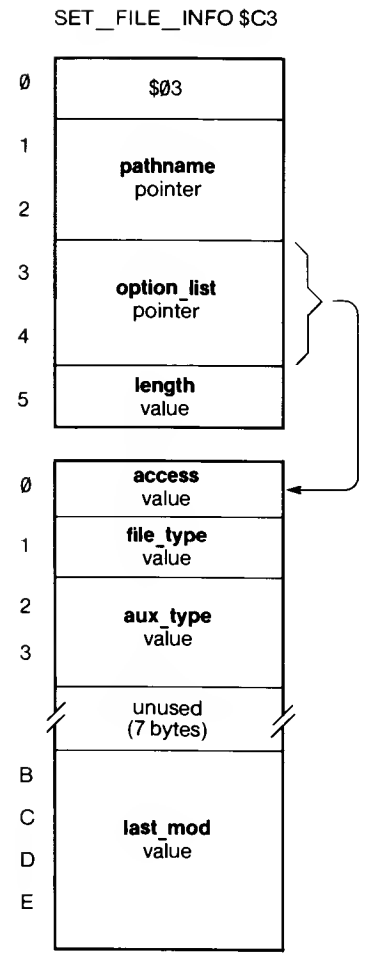
If **new_pathname** matches the pathname of an existing file, you will get a DUPERR.

Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$40:	BADPATH	Invalid pathname syntax
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	File not found
$47:	DUPERR	Duplicate file name
$4A:	CPTERR	Incompatible file format
$4B:	TYPERR	File storage type not supported
$4E:	ACCSERR	File's **access** attribute prevents RENAME
$50:	FILBUSY	File is open. Request denied.
$52:	NOTSOS	Not a SOS volume
$57:	DUPVOL	Duplicate volume
$58:	NOTBLKDEV	Not a block device

9.1.4 SET_FILE_INFO File Call $C3

This call modifies file information in the directory entry of the block file specified by the **pathname** parameter. If the file is closed, a SET_FILE_INFO call will modify the file information immediately. This information will be returned by any subsequent GET_FILE_INFO calls. If the file is open, no file information will be modified until the file is closed.



Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the file name of the file whose directory entry will be modified: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

option_list: pointer

This is a pointer to the optional parameter list if **length** is between $01 and $0F; otherwise it is ignored.

length: 1 byte value
Range: $00..$0F

This is the length of the optional parameter list. It specifies which optional parameters are supplied. If **length** equals $00, no optional parameters are supplied: the call does nothing more than error checking.

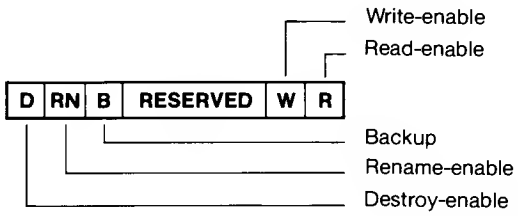
The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.

Ø = no optional parameters
1 = **access**
2 = **access** through **file_type**
4 = **access** through **aux_type**
F = **access** through **last_mod**

Optional Parameters

access: 1 byte value
Range: $ØØ..$E3
Default: None

This parameter specifies the access allowed to the file. Bits 4 through 2 are reserved for future implementation and must be set to Ø, otherwise an ACCSERR will occur.



For bits 7, 6, 1, and Ø,

Ø = not allowed
1 = allowed

These bits may be altered as the user wishes by the SET_FILE_INFO call.

For bit 5,

Ø = backup not needed
1 = backup needed

This bit is always set when a SET_FILE_INFO call is made. Only the Backup III program can clear it.

file_type: 1 byte value
Range: $ØØ..$FF
Default: Current value

This the type identifier for this file. The **file_type** does not affect the way in which SOS deals with the file: it is used only by interpreters to determine the internal arrangement and meaning of the bytes in the file. These values of **file_type** are now defined:

$ØØ = Typeless file (BASIC or Pascal "unknown" file)
$Ø1 = File containing all bad blocks on the volume
$Ø2 = Pascal or assembly-language code file
$Ø3 = Pascal text file
$Ø4 = BASIC text file; Pascal ASCII file
$Ø5 = Pascal data file
$Ø6 = General binary file
$Ø7 = Font file
$Ø8 = Screen image file
$Ø9 = Business BASIC program file
$ØA = Business BASIC data file
$ØB = Word Processor file
$ØC = SOS system file (DRIVER, INTERP, KERNEL)
$ØD, $ØE = SOS reserved
$ØF = Directory file (see **storage_type**)
$1Ø..$DF = SOS reserved
$EØ..$FF = ProDOS reserved

aux_type: 2 byte value
Range: $0000..$FFFF
Default: Current value

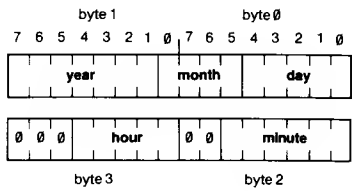
This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (**storage_type** is $0F), these bytes contain the total number of blocks on the volume.

unused: 7 bytes

These bytes are here to maintain symmetry with GET_FILE_INFO, and are always ignored by SET_FILE_INFO.

last_mod: 4 byte value
Range: $00000000..$FFFFFFFF
Default: Current value

This is the date and time the file was last closed after being written to. It can be set to a user-defined value, or you can use the GET_TIME call (see the Utility calls) and form this value from the current time. The **last_mod** parameter is organized as two 2-byte words, each stored low byte first:



The ranges for these fields are as follows:

Year:	0..99	($00..$63)
Month:	0..12	($00..$0C)
Day:	0..31	($00..$1F)
Hour:	0..24	($00..$18)
Minute:	0..60	($00..$3C)

A zero value for the month or day means that no value was set.

No checking is performed on this parameter. If you use the GET_TIME call, you must pack the 18-byte **time** parameter from that call into the proper format for the SET_FILE_INFO call's **last_mod** parameter.

Comments

The default value for all optional parameters that are omitted is the current value of that attribute of the file: for example, omitting the **last_mod** parameter results in no change to that file's modification date and time.

The same required and optional parameter lists can be used for GET_FILE_INFO. In fact, you can perform a GET_FILE_INFO, examine and perhaps alter the values in the parameter lists, and then perform a SET_FILE_INFO to update the file's attributes.

You can perform SET_FILE_INFO on any block file, regardless of the current value of its **access** attribute. In this call, therefore, an access error can result only from passing an invalid **access** parameter.

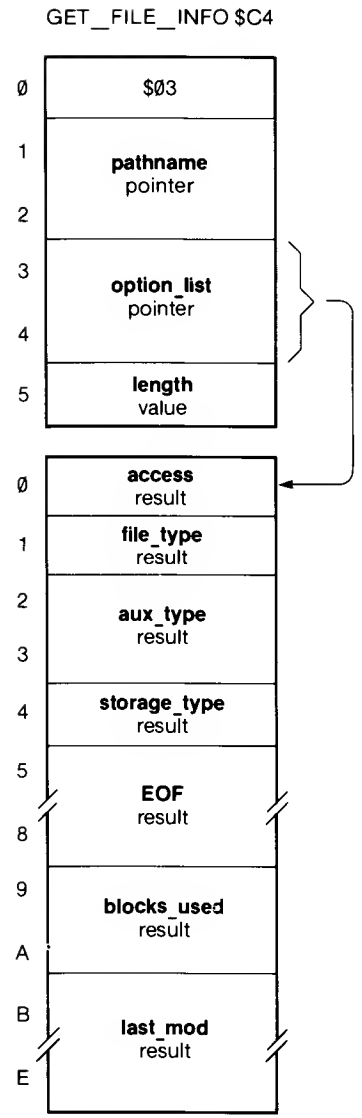
SET_FILE_INFO affects a file's directory entry only. It does not affect the FCB entry for any access path to the file. Specifically, if you open a file with read/write access, then use a SET_FILE_INFO call to change the access to read-only, you still write to the file via that access path, but you cannot open another access path. This is because the **access** field in the file's directory entry will not be updated until the file is closed, and the FCB entries will not be updated at all: so, as far as SOS is concerned, this is still a read/write file, for which only one access path is allowed. As soon as you close the file, however, the new **access** value will be stored in the directory entry, and multiple read-only access paths can be opened.

Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$40:	BADPATH	Invalid pathname syntax
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	File not found
$4A:	CPTERR	Incompatible file format
$4B:	TYPERR	Unsupported file storage type
$4E:	ACCSERR	Access parameter invalid
$52:	NOTSOS	Not a SOS volume
$53:	BADLSTCNT	Length parameter invalid
$58:	NOTBLKDEV	File is not on a block device

9.1.5 GET_FILE_INFO — File Call $C4

This call returns file information from the directory entry of the block file specified by the **pathname** parameter.



Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the pathname of the file whose directory entry information will be returned: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself.

option_list: pointer

This is a pointer to the optional parameter list if **length** is between $01 and $0F; otherwise it is ignored.

length: 1 byte value
Range: $00..$0F

This is the length of the optional parameter list. If **length** equals $00, no optional parameters are returned: the call does nothing more than error checking.

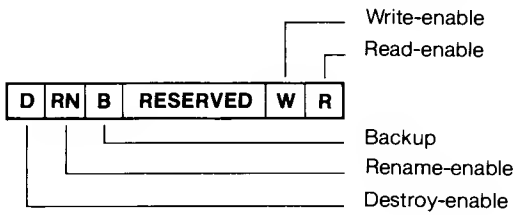
The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.

$Ø0 = no optional parameters
$Ø1 = **access**
$Ø2 = **access** through **file_type**
$Ø4 = **access** through **aux_type**
$Ø5 = **access** through **storage_type**
$Ø9 = **access** through **EOF**
$ØB = **access** through **blocks_used**
$ØF = **access** through **last_mod**

Optional Parameters

access: 1 byte result
Range: $Ø0..$C3

This parameter returns the access allowed to the file. Bits 4 through 2 are reserved for future implementation and are now set to Ø.



For bits 7, 6, 1, and Ø,

Ø = not allowed
1 = allowed

For bit 5,

Ø = backup not needed
1 = backup needed

file_type: 1 byte result
Range: $Ø0..$FF

This the type identifier for this file. The **file_type** does not affect the way in which SOS deals with the file: it is used only by interpreters to determine

the internal arrangement and meaning of the bytes in the file. These values of **file_type** are now defined:

$Ø0 = Typeless file (BASIC or Pascal "unknown" file)
$Ø1 = File containing all bad blocks on the volume
$Ø2 = Pascal or assembly-language code file
$Ø3 = Pascal text file
$Ø4 = BASIC text file; Pascal ASCII file
$Ø5 = Pascal data file
$Ø6 = General binary file
$Ø7 = Font file
$Ø8 = Screen image file
$Ø9 = Business BASIC program file
$ØB = Word Processor file
$ØC = SOS system file (DRIVER, INTERP, KERNEL)
$ØD, $ØE = SOS reserved
$ØF = Directory file (see **storage_type**)
$1Ø..$DF = SOS reserved
$EØ..$FF = ProDOS reserved

aux_type: 2 byte result
Range: $ØØØØ..$FFFF

This is the auxiliary file identifier. It is used by interpreters to store any additional information about the file. BASIC, for example, uses this field to store the record size of its data files. If the file is a volume directory (**storage_type** is $ØF), these bytes contain the total number of blocks on the volume.

storage_type: 1 byte result
Range: $Ø1..$Ø3, $ØD, $ØF

This byte describes the external format of the file: how the blocks that compose the file are stored on the volume.

$Ø1 = seedling file (Ø < = **EOF** < = 512 bytes)
$Ø2 = sapling file (512 < **EOF** < = 128K bytes)
$Ø3 = tree file (128K < **EOF** < 16M bytes)
$ØD = subdirectory file
$ØF = volume directory file

These structures are fully explained in Chapter 5. In brief, seedling files are stored as one data block; sapling files are stored as one index block and up to 256 data blocks; tree files are stored as one root index block, up to 127 subindex blocks, and up to 32,767 data blocks. Directories and subdirectories do not use index blocks, and instead are stored as doubly-linked lists of blocks.

EOF: 4 byte result
Range: $00000000..$00FFFFFF

This is the position of the end of file marker. It indicates the number of bytes readable from the file. This is the **EOF** value stored in the file's directory entry when the file was created or last closed. It is accurate only if the file is not open for writing. If the file is open for writing, the current **EOF** (stored in the file's FCB entry) can be read by the GET_EOF call.

blocks_used: 2 byte result
Range: $0000..$FFFF

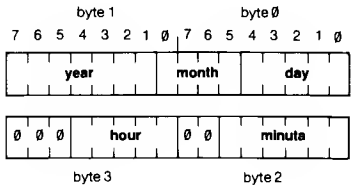
If the file is a standard file or subdirectory (**storage_type** is $01, $02, $03, or $0D), **blocks_used** is the total number of blocks (including index blocks) currently used by the file.

If the file is a sparse file, the **blocks_used** value can be substantially less than one would expect from the **EOF**.

If the file is a volume directory (**storage_type** is $0F), **blocks_used** is the total number of blocks used by all files on the volume.

last_mod: 4 byte result
Range: $00000000..$FFFFFFFF

This is the date and time the file was last closed after being written to. If the file has never been written to, these bytes are the same as the creation date of the file. SET_FILE_INFO can also change the modification date.



The ranges for these fields are as follows:

Year:	0..99 ($00..$63)
Month:	0..12 ($00..$0C)
Day:	0..31 ($00..$1F)
Hour:	0..24 ($00..$18)
Minute:	0..60 ($00..$3C)

A zero value for the month or day means that no value was set.

Comments

This call can be performed when the file is either open or closed. The same required and optional parameter lists can be used for SET_FILE_INFO. A GET_FILE_INFO call to an open file will return file information from the directory entry, not access path information from the FCB entry. This is not surprising, since the GET_FILE_INFO call refers to a file by its **pathname**, not its **ref_num**. For example, if you have changed the **EOF** since the file was opened, GET_FILE_INFO will not return the current value.

Errors

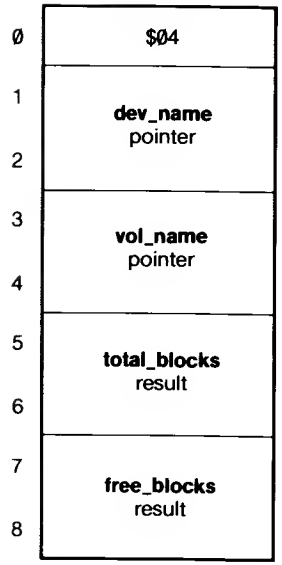
$27:	IOERR	I/O error
$40:	BADPATH	Invalid pathname syntax
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	File not found
$4A:	CPTERR	Incompatible file format
$4B:	TYPERR	Unsupported file storage type
$52:	NOTSOS	Not a SOS volume
$53:	BADLSTCNT	Length parameter invalid
$58:	NOTBLKDEV	Not a block device

9.1.6 VOLUME

File Call $C5

When given the name of a device, this call returns the volume name of the volume contained in that device, the number of blocks on that volume, and the number of currently unallocated blocks on that volume.

VOLUME $C5



Required Parameters

dev_name: pointer

This parameter is a pointer to a string containing the device name: the first byte of the string contains the number of bytes in the device name; the remaining bytes contain the device name itself.

vol_name: pointer

This is a pointer to a buffer at least $1Ø bytes long into which the volume name will be returned: the first byte in the buffer contains the number of bytes in the volume name; the rest contain the name itself.

total_blocks: 2 byte result
Range: $ØØØØ..$FFFF

This is the total number of blocks contained by the volume in the specified block device.

free_blocks: 2 byte result
Range: $ØØØØ..$FFFF

This is the number of unallocated blocks contained by the volume in the specified block device.

Comments

The **dev_name** must point to the name of a block device.

Errors

$1Ø:	DNFERR	Device not found
$27:	IOERR	I/O error
$45:	VNFERR	Volume not found
$4A:	CPTERR	Incompatible file format
$52:	NOTSOS	Not a SOS volume
$58:	NOTBLKDEV	Not a block device

9.1.7 SET_PREFIX File Call $C6

This call sets the current SOS prefix pathname to that specified by **pathname**.

SET_PREFIX $C6

0	$01
1	**pathname** pointer
2	

Required Parameters

pathname: pointer

This parameter is a pointer to a string containing the pathname that will replace the current prefix pathname: the first byte of the string contains the number of bytes in the pathname; the remaining bytes contain the pathname itself. This pathname specifies a volume directory or subdirectory, not a character file or a standard file.

Comments

The system prefix is appended to the beginning of any pathname not beginning in a volume name or device name: a volume name is preceded by a slash, and a device name begins with a period.

If the new prefix begins with a volume name, only syntax checking is performed on it: SOS does not verify that the directory file specified by the prefix is actually on line. If the new prefix begins with a device name, SOS substitutes the corresponding volume name: the SOS prefix always begins with a volume name.

The prefix can be reset to null by passing a pathname with a length of zero characters.

Upon system boot, the prefix is initialized to the volume directory name of the boot disk.

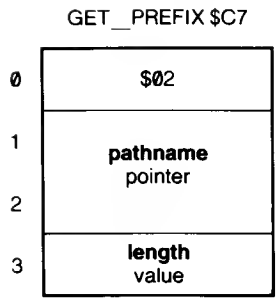
The **pathname** can optionally terminate with a "/".

Errors

$27:	IOERR	I/O error
$40:	BADPATH	Invalid pathname syntax
$58:	NOTBLKDEV	Not a block device

9.1.8 GET_PREFIX File Call $C7

This call returns the current SOS prefix pathname.



Required Parameters

pathname: pointer

This parameter is a pointer to a string into which SOS is to store the current prefix pathname: the first byte of the string contains the number of bytes in the prefix; the remaining bytes contain the prefix itself.

length: 1 byte value
Range: $00..$FF
Default: $80

This is the maximum number of bytes in the **pathname** buffer. This should be set as long as the longest prefix the interpreter accepts: SOS will accept up to 128 ($80) bytes. A BTSERR is returned if the pathname is longer than **length**.

Comments

If the SOS prefix pathname has been set to the null string (no prefix), the null string is returned.

If the prefix pathname is not null, it is terminated with a slash.

If the first name in the prefix pathname is a volume name, the pathname begins with a slash.

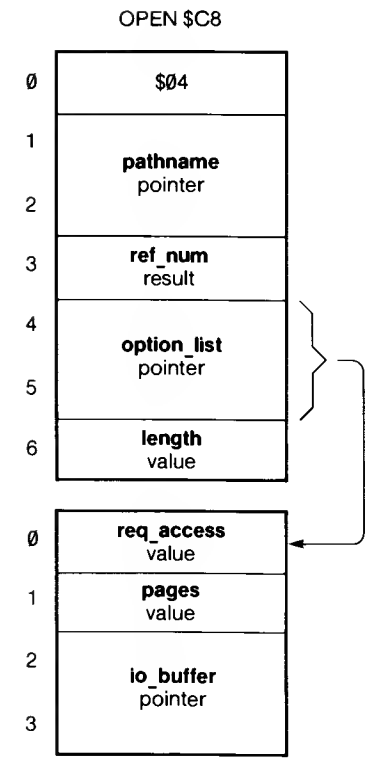
Errors

$4F:	BTSERR	Buffer too small

9.1.9 OPEN

File Call $C8

This call causes SOS to open an access path (allowing read-access, write-access, or both) to the file specified by **pathname**. For this access path, SOS makes an entry in the file control block and allocates a 1Ø24-byte I/O buffer. This buffer holds the contents of one index block (if the file has any) and one data block.



Required Parameters

pathname: pointer

This is a pointer to a string in memory containing the pathname of the file to be opened: the first byte is the number of characters in the pathname; the remaining bytes are the characters of the pathname itself. It may be any block or character file.

ref_num: 1 byte result
Range: $Ø1..$1Ø, $81..$9Ø

The reference number is assigned when an access path to a file is opened. It uniquely identifies an access path to the file: any open-file call will operate on a single access path, not the file itself.

option_list: pointer

This points to optional parameter list if **length** is between $Ø1 and $Ø4; otherwise it is ignored.

length: 1 byte value
Range: $ØØ..$Ø4

This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.

The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.

$ØØ = no optional parameters
$Ø1 = **req_access**
$Ø4 = **req_access** through **io_buffer**

Optional Parameters

req_access: 1 byte value
Range: $ØØ..$Ø3
Default: $ØØ

This is the requested file access. SOS compares this parameter with the file's current access-attribute byte to ensure that the intended file operations are permitted. A $ØØ requests as much access as permitted.

$ØØ = Open as permitted
$Ø1 = Open for reading only
$Ø2 = Open for writing only
$Ø3 = Open for reading and writing

A standard file that is already open for writing may have only one access path: a **req_access** of $ØØ will open the existing access path for reading as well. A standard file on a write-protected volume may never be opened for writing; a **req_access** of $ØØ will open such a file for reading only.

A character file may have multiple access paths with read-access, write-access, or both, if the file's device allows such access.

pages: 1 byte value
Range: $ØØ or $Ø4
Default: $ØØ

This is the length in 256-byte pages of a caller-supplied I/O buffer. If equal to $ØØ, then SOS finds its own buffer, ignoring the **io_buffer** parameter below. If equal to $Ø4, then SOS will use the 1Ø24-byte buffer pointed to by **io_buffer**. Any value except $ØØ or $Ø4 is invalid.

If **pages** is nonzero, you must specify an **io_buffer** parameter.

In general, it is preferable to let SOS allocate an I/O buffer.

io_buffer: pointer

This is an indirect pointer to a caller-supplied I/O buffer if and only if the **pages** parameter is nonzero.

Comments

On block files, multiple access paths for read-access are permitted.

On block files, only one access path for writing is permitted: no other access path, even for reading only, is permitted at the same time.

Multiple access paths on character files for both read- and write-access are permitted.

OPEN sets the file **level** of the opened file to the current system file level (see SET_LEVEL and GET_LEVEL). Unless the file level is raised, a subsequent CLOSE or FLUSH of multiple files will close or flush this file.

The **option_list** and **length** parameters are ignored when OPENing character files; no optional parameters are used.

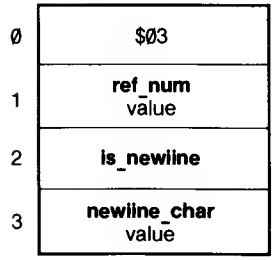
Errors

$27:	IOERR	I/O error
$40:	BADPATH	Invalid pathname syntax
$41:	CFCBFULL	Character File Control Block table full
$42:	FCBFULL	Block File Control Block table full
$44:	PNFERR	Path not found
$45:	VNFERR	Volume not found
$46:	FNFERR	File not found
$4A:	CPTERR	Incompatible file format
$4B:	TYPERR	Unsupported file storage type
$4E:	ACCSERR	File doesn't allow this **req_access**
$4F:	BTSERR	User-supplied buffer too small
$50:	FILBUSY	Can't open for multiple writes
$52:	NOTSOS	Not a SOS diskette
$53:	BADLSTCNT	Length parameter invalid
$54:	OUTOFMEM	Out of free memory for buffer
$55:	BUFTBLFULL	Buffer table full
$56:	BADSYSBUF	Invalid system buffer parameter
$57:	DUPVOL	Duplicate volume

9.1.10 NEWLINE File Call $C9

This call allows the caller to turn newline read mode on or off. Once newline mode has been turned on, any subsequent READ operation will immediately terminate if the newline character is encountered in the input byte stream.

NEWLINE $C9



Required Parameters

ref_num: 1 byte value
Range: $01..$10, $81..$90

This is the reference number of the access path, provided by the OPEN call.

is_newline: 1 byte value
Range: $00..$FF

The high bit of this byte determines whether newline read mode is on or off. If it is set (**is_newline** > $7F), newline mode is on; otherwise, newline mode is off.

newline_char: 1 byte value
Range: $00..$FF

This byte indicates the character used to terminate read requests. If newline read mode is off, this parameter is ignored.

Comments

The **newline_char** byte need not have any ASCII interpretation.

A NEWLINE call to a character file implicitly does a D_CONTROL call number 2 (set newline mode) to the device driver represented by that file. This changes the newline mode of all access paths to that character file.

Errors

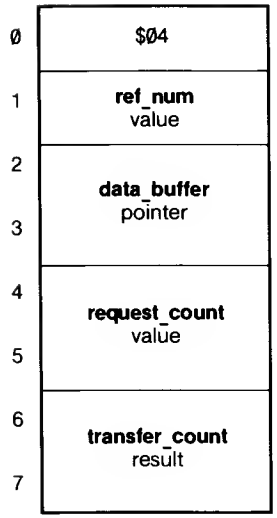
$43:	BADREFNUM	Bad reference number

9.1.11 READ

File Call $CA

This call attempts to transfer **request_count** bytes, starting from the current file position (**mark**), from the I/O buffer of the file access path specified by **ref_num** into the interpreter's data buffer pointed to by **data_buffer**. If newline read mode is enabled and the newline character is encountered before **request_count** bytes have been read, then the **transfer_count** parameter will be less than **request_count** and exactly equal to the number of bytes transferred, including the newline byte.

READ $CA



Required Parameters

ref_num: 1 byte value
Range: $01..$10, $81..$90

This is the reference number of the access path to be read from, obtained through an OPEN call.

data_buffer: pointer

This is a pointer to the first byte of a caller-supplied buffer at least **request_count** bytes long.

request_count: 2 byte value
Range: $0000..$FFFF

This is the number of bytes SOS is to read from the file into the buffer. If **request_count** equals $0000, the READ call does error checking only: no bytes are read.

transfer_count: 2 byte result
Range: $0000..**request_count**

If a READ is successful, the number of bytes transferred to the data buffer is returned in this parameter. If a READ is completely unsuccessful, **transfer_count** equals $0000.

Comments

READ advances the current file position (**mark**) by one byte for each byte transferred. It will advance the **mark** up to the end-of-file (**EOF**) marker, which points one byte past the last byte in the file. READ fails with an EOFERR if and only if the **mark** already equals **EOF**; in this case, no bytes are transferred and **transfer_count** returns zero.

If a READ operation spans several contiguous blocks on a disk, SOS transfers whole blocks directly to the interpreter's data buffer, bypassing the I/O buffer; partial blocks go through the I/O buffer. This optimization improves performance, but is otherwise invisible to the interpreter writer and user.

Errors

$27:	IOERR	I/O error
$43:	BADREFNUM	Invalid reference number
$4C:	EOFERR	End of file has been encountered
$4E:	ACCSERR	File not open for READing

9.1.12 WRITE File Call $CB

This call attempts to transfer **request_count** bytes, starting from the current file position (**mark**), from the buffer pointed to by **data_buffer** to the open file specified by **ref_num**.

WRITE $CB

0	$03
1	**ref_num** value
2	**data_buffer** pointer
3	
4	**request_count** value
5	

Required Parameters

ref_num: 1 byte value
Range: $01..$10, $81..$90

This is the reference number of the file to be written to, obtained by an OPEN call.

data_buffer: pointer

This is a pointer to a caller-supplies buffer from which SOS is to draw the bytes to be written to the file. This pointer is not modified by SOS.

request_count: 2 byte value
Range: $0000..$FFFF

This is the number of bytes to be written to the file.

Comments

If WRITE ends with an OVRERR, it has written all the bytes that it can to the file: it will not tell you how many it has written. Otherwise, WRITE always succeeds or fails completely.

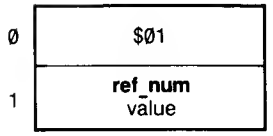
Bytes written to a file may be stored in an I/O buffer, and sent a buffer-load at a time. For block files, WRITE physically alters the bytes on the volume only when a block of bytes has been written to the file: this occurs automatically when the **mark** crosses a block boundary. To ensure that information in the buffer has been updated on the volume, use the FLUSH call.

Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume write-protected
$43:	BADREFNUM	Invalid reference number
$48:	OVRERR	Not enough room in file or on volume
$4E:	ACCSERR	Tried to write to read-only file

9.1.13 CLOSE File Call $CC

CLOSE $CC



The file access path specified by **ref_num** is closed. Its file control block (FCB) entry is deleted, and if the file is a block file that has been written to, its I/O buffer is written to the file. The directory entry of a block file is then updated from the FCB entry. Further file operations using that **ref_num** will fail.

Required Parameters

ref_num: 1 byte value
Range: $00..$10, $81..$90

This is the reference number of the file to be closed, obtained by an OPEN call.

Comments

If a block file has been written to, a CLOSE call changes the modification date and time of the file to the current date and time.

If **ref_num** equals $00, all open files are closed whose file **level** (see SET_LEVEL, GET_LEVEL) is greater than or equal to the current system level.

If an error occurs while closing multiple files, all files that can be closed will be, and CLOSE will return the error number of the last error that occurred. CLOSE will not tell you which files were closed and which were not.

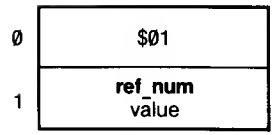
Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$43:	BADREFNUM	Invalid reference number
$48:	OVRERR	Not enough room on volume

9.1.14 FLUSH — File Call $CD

If a previous WRITE call has left any data in a block file's I/O buffer, the FLUSH call writes these data to the volume the file is stored on and clears the buffer. If the I/O buffer is empty, FLUSH simply returns an error code of $00.

FLUSH $CD



Required Parameters

ref_num: 1 byte value
Range: $00..$10

This is the reference number of the block file access path to be FLUSHed, obained from an OPEN call. Since the file is open for writing, this access path is the only one.

Comments

FLUSH must be used only on block file access paths that are open for writing.

If the **ref_num** equals $00, all open files are FLUSHed whose file **level** (see SET_LEVEL, GET_LEVEL) is greater than or equal to the current system file level.

FLUSH is a time-consuming call: if it is used when not needed, performance will suffer.

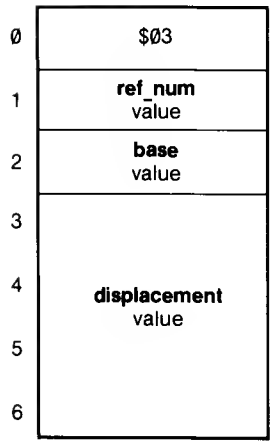
Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume is write-protected
$43:	BADREFNUM	Invalid reference number
$48:	OVRERR	Not enough room on volume
$58:	NOTBLKDEV	Not a block device

9.1.15 SET_MARK File Call $CE

This call changes the current file position (**mark**) of the file access path specified by **ref_num**. The **mark** can be changed to an absolute byte position in the file, or to a position relative to the **EOF** or the current **mark**.

SET_MARK $CE



Required Parameters

ref_num: 1 byte value
Range: $01..$10

This is the reference number of the block file access path whose **mark** is to be moved, obtained through an OPEN call.

base: 1 byte value
Range: $00..$03

This is the starting byte position in the file from which to calculate the new **mark** position.

$00 = Absolute, byte $00000000..$00FFFFFF
$01 = Backward from **EOF**
$02 = Forward from current **mark**
$03 = Backward from current **mark**

displacement: 4 byte value
Range: $00000000..$00FFFFFF

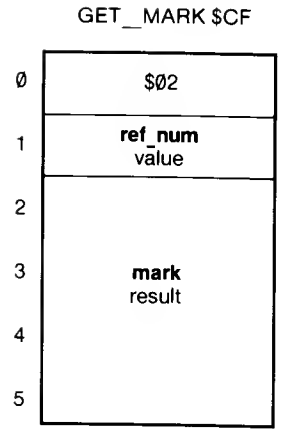
This is the number of bytes the **mark** is to move from the starting location specified by the **base** parameter. The final computed position must lie between $0 and the current **EOF** ($0 <= **mark** <= **EOF** <= $FFFFFF).

Errors

$27:	IOERR	I/O error
$43:	BADREFNUM	Invalid reference number
$4D:	POSNERR	Position out of range
$58:	NOTBLKDEV	Not a block device

9.1.16 GET_MARK File Call $CF

This call returns the current file position (**mark**) of the block file access path specified by **ref_num**.



Required Parameters

ref_num1: 1 byte value
Range: $01..$10

This is the reference number of the file whose current position is to be returned.

mark: 4 byte result
Range: $00000000 through
current **EOF** value

This is the current **mark** position in the file.

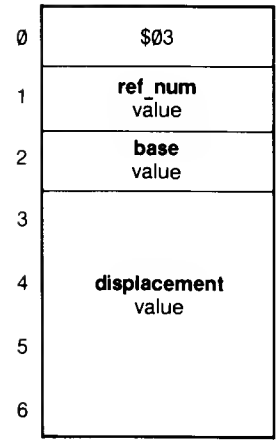
Errors

$43:	BADREFNUM	Invalid reference number
$58:	NOTBLKDEV	Not a block device

9.1.17 SET_EOF

File Call $DØ

SET_EOF $DØ



This call changes the end-of-file marker (**EOF**) of the block file whose access path is specified by **ref_num**. The **EOF** can be changed to an absolute byte position, or to a position relative to the current **EOF** or the current **mark**.

If the new **EOF** is less than the current **EOF**, empty blocks at the end of the file are released to the system and their data are lost. If the new **EOF** is greater than the current **EOF**, blocks are not physically allocated for unwritten data. (This is one way of creating a sparse file.) If a program attempts to read from these newly created logical positions before they have been physically written to, SOS supplies a null ($ØØ) for each byte requested.

Required Parameters

ref_num: 1 byte value
Range: $Ø1..$1Ø

This is the reference number of the file whose **EOF** is to be changed, returned by an OPEN call. It must refer to a block file open for writing, and is thus the file's sole **ref_num**.

base: 1 byte value
Range: $00..$03

This is the position in the file from which to calculate the new value of **EOF**, (the current number of bytes in the file).

$00 = Absolute, byte $000000..$FFFFFF
$01 = Backward from current **EOF**
$02 = Forward from current **mark** position
$03 = Backward from current **mark** position

displacement: 4 byte value
Range: $00000000..$00FFFFFF

This is the number of bytes the **EOF** is to move from the starting position specified in the **base** parameter. The final computed position must be greater than or equal to $000000, and less than or equal to $FFFFFF.

Comments

The file must be a block file currently open for writing. Since such a file can have only one access path, the **ref_num** specifies the file, as well as the access path.

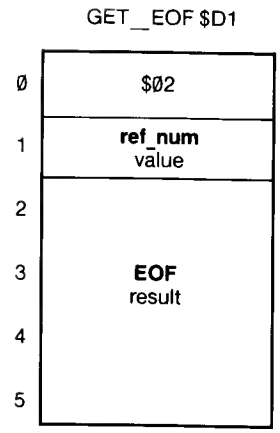
This call updates the **EOF** field in the file control block entry, but not the **EOF** field in the file's directory entry: the latter is updated only when the access path is closed. For this reason, a GET_FILE_INFO call to an open file will not always return the current **EOF**. A GET_EOF call will.

Errors

$27:	IOERR	I/O error
$2B:	NOWRITE	Volume write-protected
$43:	BADREFNUM	Invalid reference number
$4D:	POSNERR	Position out of range
$4E:	ACCSERR	Tried to move **EOF** of read-only file
$58:	NOTBLKDEV	Not a block device

9.1.18 GET_EOF — File Call $D1

This returns the current end-of-file (**EOF**) position of the file specified by **ref_num**.



Required Parameters

ref_num: 1 byte value
Range: $01..$10

This is the reference number of the file whose current position is to be returned, provided by an OPEN call.

EOF: 4 byte result
Range: $00000000..$00FFFFFF

This is the number of bytes that can be read from the file.

Errors

$43:	BADREFNUM	Invalid reference number
$58:	NOTBLKDEV	Not a block device

9.1.19 SET_LEVEL File Call $D2

This call changes the current value of the system file level. All subsequent OPENs will assign this level to the files opened. All subsequent CLOSE and FLUSH operations on multiple files (using a **ref_num** of $ØØ) will operate on only those files that were opened with a **level** greater than or equal to the new level.

SET_LEVEL $D2

Ø	$Ø1
1	**level** value

Required Parameters

level: 1 byte value
Range: $Ø1..$Ø3

This specifies the new file level.

Comments

The system file level is set to $Ø1 at boot time.

Errors

$59: LEVLERR Invalid file level

9.1.20 GET_LEVEL — File Call $D3

This call returns the current value of the system file level. See SET_LEVEL, OPEN, CLOSE, and FLUSH.

GET_LEVEL $D3

0	$01
1	**level** result

Required Parameters

level: 1 byte result
Range: $01..$03

This parameter returns the current file level.

Comments

The file level is set to $01 at boot time.

9.2 File Call Errors

These error messages can be generated by SOS file calls; in addition, some of these calls may generate device call errors, described in section 10.2. Other errors are listed in Appendix D.

$40: Invalid pathname syntax (BADPATH)

The pathname violates the syntax rules in Chapter 4 of Volume 1.

$41: Character File Control Block full (CFCBFULL)

The Character File Control Block (CFCB) table can contain a maximum of $10 entries. Opening the same character file more than once will return the same **ref_num** (that is, will not consume an additional entry).

$42: Block File or Volume Control Block full (FCBFULL)

The Block File Control Block (BFCB) table can contain a maximum of $1Ø entries. The Volume Control Block (VCB) table can contain a maximum of $Ø8 entries. Opening the same block file more than than once returns a different **ref_num** and consumes a new entry in the BFCB table. Every volume with an open file on it, whether it is mounted on a device or not, consumes one entry in the VCB table.

$43: Invalid reference number (BADREFNUM)

The **ref_num** input parameter does not match the **ref_num** of any currently open file. This error is also returned if the currently open file is marked with a bad **storage_type**; only $Ø1 through $Ø4, $ØD, and $ØF are allowed.

$44: Path not found (PNFERR)

Some file name in the pathname refers to a nonexistent file. The pathname's syntax is legal.

$45: Volume not found (VNFERR)

The volume name in the pathname refers to a nonexistent volume directory. The pathname's syntax is legal.

$46: File not found (FNFERR)

The last file name in the pathname refers to a nonexistent file. The pathname's syntax is legal. Note that a missing volume directory file returns VNFERR instead of FNFERR.

$47: Duplicate file name (DUPERR)

An attempt was made to CREATE a file using a **pathname** that already belongs to a file, or a RENAME was attempted using a **new_pathname** that already belongs to a file.

$48: Overrun on volume (OVRERR)

An attempt to allocate blocks on a volume during a CREATE or WRITE operation failed due to lack of space on the volume. This error also is returned on an invalid **EOF** parameter.

$49: Directory full (DIRFULL)

No more entries are left in the root/subdirectory. Thus no more files can be added (CREATEd) in this directory until another file is DESTROYed.

$4A: Incompatible file format (CPTERR)

The file is not backward compatible with this version of SOS.

$4B: Unsupported storage type (TYPERR)

The CREATE call accepts only two values for the **storage_type** parameter: $Ø1 (standard file) or $ØD (subdirectory file).

$4C: End of file would be exceeded (EOFERR)

A READ call was attempted when the **mark** was equal to the **EOF**.

$4D: Position out of range (POSNERR)

A base/displacement parameter pair produced an invalid **mark** or **EOF**.

$4E: Access not allowed (ACCSERR)

The user attempted to access (RENAME, DESTROY, READ from, or WRITE to) a file in a way not allowed by its **access** attribute.

$4F: Buffer too small (BTSERR)

The user supplied a buffer too small for its purpose.

$5Ø: File busy (FILBUSY)

An attempt was made to RENAME or DESTROY an open file or to OPEN a block file already open for writing.

$51: Directory error (DIRERR)

The directory entry count disagrees with the actual number of entries in the directory file.

$52: Not a SOS volume (NOTSOS)

The volume in the block device contains a directory that is not in SOS format: it may be an Apple II Pascal or DOS 3.3 volume.

$53: Length parameter invalid (BADLSTCNT)

The **length** supplied for the optional parameter list is invalid.

$54: Out of memory (OUTOFMEM)

There is not enough free memory for the SOS system buffer. The user must release some memory to SOS to allow the system to use it.

$55: Buffer Table full (BUFTBLFULL)

The Buffer Table can contain a maximum of $1Ø entries.

$56: Invalid system buffer parameter (BADSYSBUF)

The buffer pointer parameter must be an extended indirect pointer.

$57: Duplicate volume (DUPVOL)

A SOS call asked SOS to bring a volume on-line on a particular block device. The request was denied because a volume with the same name on another block device is currently on line and contains a currently open file.

$58: Not a block device (NOTBLKDEV)

Only OPEN, NEWLINE, READ, WRITE, and CLOSE file calls can reference a character file. For example, CREATE is not permitted on the character file .PRINTER .

$59: Invalid level (LVLERR)

The SET_LEVEL call received a parameter less than $Ø1 or greater than $Ø3.

$5A: Invalid bit map address (BITMAPADR)

An index block contained a block number that, according to the bit map, is not physically available on the volume: usually this indicates that the blocks on the volume have been scrambled.

10

Device Calls and Errors

Device Calls

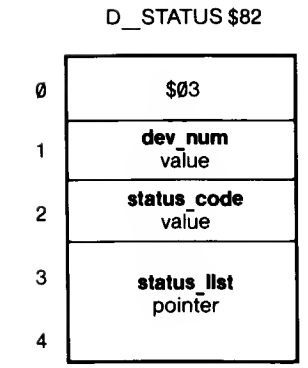
These SOS calls operate directly on devices.

$82: D_STATUS
$83: D_CONTROL
$84: GET_DEV_NUM
$85: D_INFO

10.1.1 D_STATUS

Device Call $82

This call returns status information about a particular device. The information can be either general or device-specific information. D_STATUS returns information about the internal status of the device or its driver; GET_DEV_INFO returns information about the external status of the driver and its interface with SOS.



Required Parameters

dev_num: 1 byte value
Range: $01..$18

This is the device number of the device from which to read status information, obtained from the GET_DEV_NUM call. Each device in the system has a unique device number assigned to it when the system is booted. Device numbers do not change unless the SOS.DRIVER file is changed and the system is rebooted.

status_code: 1 byte value
Range: $00..$FF

This is the number of the status request being made. All device drivers respond to the following requests:

Block devices only:

$00 Return driver's status byte

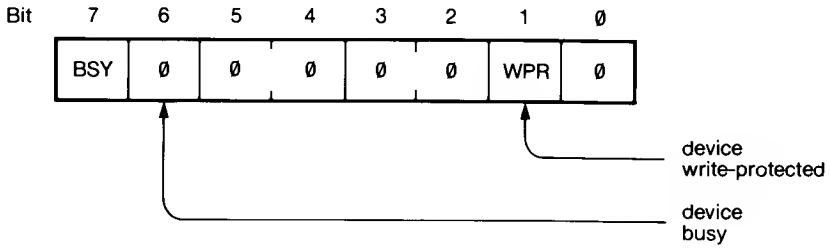
Character devices only:

$00 No effect
$01 Return driver's control block
$02 Return newline status

Device drivers also may respond to other status codes. The complete list of status requests available for a device driver is included in the documentation accompanying that driver.

status_list: pointer

This is a pointer to the buffer in which the device driver returns its status. For the three requests above, the buffer is in one of these three formats:



Bit	Meaning
BSY	If 1, device is busy
WPR	If 1, device or medium is write-protected

Figure 10-1. Block Device Status Request $00

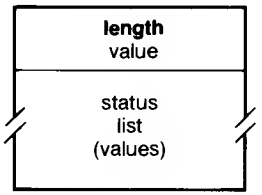


Figure 10-2. Character Device Status Request $01

The status list for each driver has a different format. See the manual describing that driver.

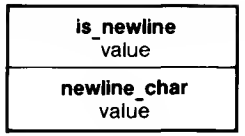


Figure 10-3. Character Device Status Request $02

The newline character is called the *termination character* in the *Apple III Standard Device Drivers Manual*.

Each driver that defines its own additional status requests also defines buffer formats for those requests; see the manual describing that driver.

Comments

The length of the buffer pointed to by **status_list** must vary depending upon the particular status request being made.

Errors

$11:	BADNUM	Invalid device number
$21:	CTLCODE	Invalid status code
$23:	NOTOPEN	Character device not open
$25:	NORESRC	Resource not available
$30..$3F		Device-specific error

10.1.2 D_CONTROL Device Call $83

This call sends control information to a particular device. The information can be either general or device-specific information. D_CONTROL operates on character devices only.

D_CONTROL $83

0	$03
1	**dev_num** value
2	**control_code** value
3	**control_list** pointer
4	

Required Parameters

dev_num: 1 byte value
Range: $Ø1..$18

This is the device number of the device to which to send control information, obtained from the GET_DEV_NUM call. Each device in the system has a unique device number assigned to it when the system is booted. Device numbers do not change unless the SOS.DRIVER file is changed and the system is rebooted.

control_code: 1 byte value
Range: $ØØ..$FF

This is the number of the control request being made. All character device drivers respond to the following requests:

$ØØ Reset device
$Ø1 Restore driver's control block
$Ø2 Set newline mode and character

Block devices do not respond to any control requests.

Device drivers also may respond to other control requests. The complete list of control requests available for a device driver is included in the documentation accompanying that driver.

control_list: pointer

This is a pointer to the buffer from which the device driver draws the control information. For the two requests above, the buffer is in one of these two formats:

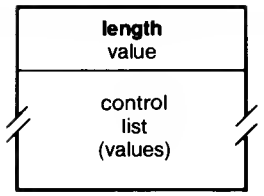


Figure 10-4. Character Device Control Code $Ø1

The status list for each driver has a different format. See the manual describing that driver.

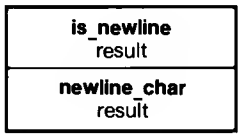


Figure 10-5. Character Device Control Code $Ø2

The newline character is called the *termination character* in the *Apple III Standard Device Drivers Manual*.

Each driver that defines its own additional control requests also defines buffer formats for those requests; see the documentation for that driver.

Comments

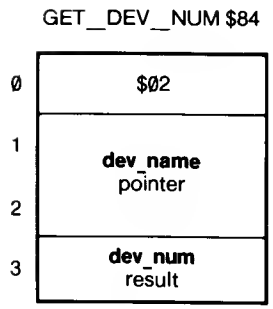
The length of the buffer pointed to by **control_list** must vary depending upon the particular control request being made.

Errors

$11:	BADNUM	Invalid device number
$21:	CTLCODE	Invalid control code
$23:	NOTOPEN	Character device not open
$25:	NORESRC	Resource not available
$26:	BADOP	No control of block devices allowed
$3Ø..$3F		Device-specific error

10.1.3 GET_DEV_NUM Device Call $84

This call returns the device number of the driver whose device name is specified. The device need not be open. The **dev_num** returned is used in the D_STATUS, D_CONTROL, and D_INFO calls.



Required Parameters

dev_name: pointer

This is a pointer to a string in memory containing the device name of the device whose number is to be returned: the first byte of the string is the number of bytes in the name; the rest are the bytes of the name itself. Note that this a device name, not a pathname.

dev_num: 1 byte result
Range: $Ø1..$18

This is the device number of the device specified by **dev_name**. The name of a device can be changed by the System Configuration Program.

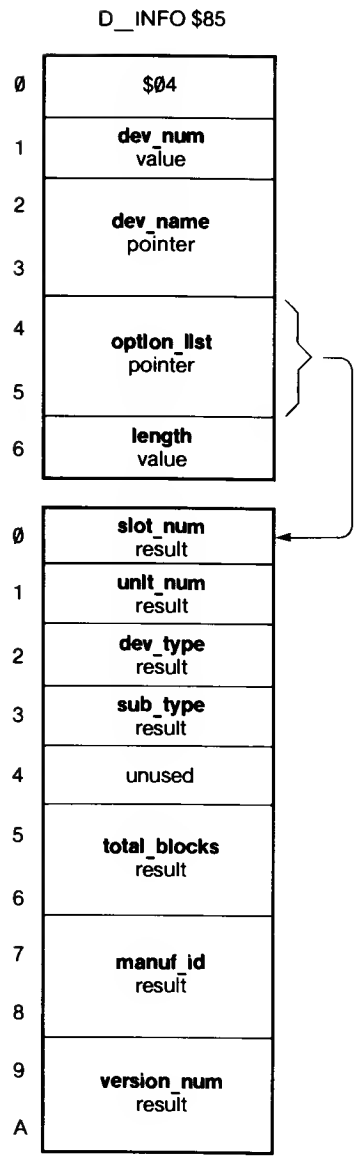
Errors

$1Ø:	DNFERR	Device name not found

10.1.4 D_INFO

Device Call $85

This call returns the device name (and optionally, other information) about the device specified by **dev_num**. The device's character file need not be open. D_INFO returns identifying information about the device's external status and interface to SOS; D_STATUS returns information about the internal status of the device and its driver.



Required Parameters

dev_num: 1 byte value
Range: $Ø1..$18

This is the device number of the device whose information is to be returned, obtained from the GET_DEV_NUM call.

dev_name: pointer

This is a pointer to a sixteen-byte buffer into which SOS is to store the resulting device name: the first byte of the buffer is the number of bytes in the name; the rest are the bytes of the name itself.

option_list: pointer

This is a pointer to the optional parameter list if **length** is between $ØØ and $ØA; otherwise it is ignored.

length: 1 byte value
Range: $ØØ..$ØA

This is the length in bytes of the optional parameter list. It specifies which optional parameters are supplied.

The values below tell the number of bytes in a list with complete parameters. If SOS receives an intermediate value, it does not take half a parameter, but reduces the **length** to the next defined value.

$ØØ = no optional parameters
$Ø1 = **slot_num**
$Ø2 = **slot_num** through **unit_num**
$Ø3 = **slot_num** through **dev_type**
$Ø5 = **slot_num** through **sub_type**
$Ø7 = **slot_num** through **total_blocks**
$Ø9 = **slot_num** through **manuf_id**
$ØB = **slot_num** through **version_num**

Optional Parameters

slot_num: 1 byte result
Range: $ØØ..$Ø4

This is the slot number of the peripheral slot the device uses. Slot numbers $Ø1 through $Ø4 correspond to peripheral slots 1 through 4. Slot number $ØØ indicates the device does not use a peripheral slot.

unit_num: 1 byte result
Range: $ØØ..$FF

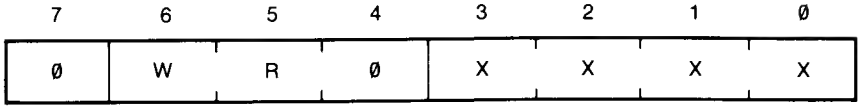
This is the unit number of the device. Devices that are bundled together into one driver module are assigned unit numbers in ascending sequence, beginning with $ØØ. See the *Apple III SOS Device Driver Writer's Guide* for more details.

This parameter has nothing to do with the logical unit numbers that Pascal associates with the devices.

dev_type: 1 byte result
Range: $ØØ..$FF

The **dev_type** byte, along with the following byte, is used for device classification and identification. This field specifies the generic family that the device belongs to.

The **dev_type** byte for SOS character devices has the following structure:



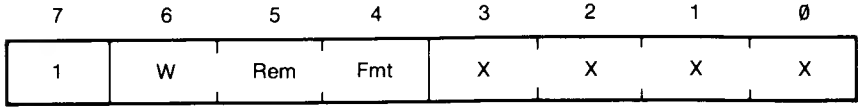
Bit 7 is cleared for all character devices.

Bit 6 (W) is *write allowed* byte. It must be set for all character devices that accept data from the Apple III.

Bit 5 (R) is the *read allowed* bit. It must be set for all character devices that send data to the Apple III.

Bit 4 is reserved for future use and must always be cleared.

The **dev_type** byte for SOS block devices has the following structure:



Bit 7 is set for all block devices.

Bit 6 (W) is *write allowed* byte. It must be set for all block devices that accept data from the Apple III.

Bit 5 (R) is the *removable device* bit. It must be set for all block devices that use removable storage media, such as floppy-disk drives.

Bit 4 is set if the driver can also format its device.

sub_type: 1 byte result
Range: $ØØ..$FF

The device subtype identifies the specific device within the generic family specified in **dev_type**.

unused: 1 byte

total_blocks: 2 byte result
Range: $0000..$FFFF

If the device is a block device, this parameter indicates the total number of blocks it can access. If the device is a character device, this parameter returns $0000. The Apple III's built-in disk drive can access $0118 blocks.

manuf_id: 2 byte result
Range: $0000..$FFFF

The manufacturer identification code uniquely identifies the manufacturer of the driver. The currently assigned values are

$0000 Unknown
$0001 Apple Computer,Inc.

version_num: 2 byte result
Range: $0000..$9999

This is the version number of the device driver. The format is BCD (binary-coded decimal); no hexadecimal digits from $A to $F will appear in this result.

Comments

The following values for **dev_type** and **sub_type** are assigned:

dev_name	dev_type	sub_type
RS232 printer (.PRINTER)	$41	$01
Silentype printer (.SILENTYPE)	$41	$02
Parallel printer (.PARALLEL)	$41	$03
Sound port (.AUDIO)	$43	$01
System console (.CONSOLE)	$61	$01
Graphics screen (.GRAFIX)	$62	$01
Onboard RS232 (.RS232)	$63	$01
Parallel card (.PARALLEL)	$64	$01
Disk III (.D1 through .D4)	$E1	$01
ProFile disk (.PROFILE)	$D1	$02
Block device formatter:		
Disk III (.FMTD1 ... .FMTD4)	$11	$01

Please contact the PCS Division Product Support Department of Apple Computer, Inc. if you wish to be assigned a **dev_type**, **sub_type**, **manuf_id**, or **version_num**. This will ensure that such codes are unique and are known to SOS and future application programs.

Errors

$11: BADNUM Invalid device number

10.2 Device Call Errors

The errors below are generated by SOS device calls; some of them are also generated by SOS file calls. Other errors are listed in Appendix D.

$10: Device not found (DNFERR)

The device name passed as a parameter to GET_DEV_NUM is not that of a device that is configured into the system: a device driver with that name was not in the SOS.DRIVER file at the time the system was booted, or that device driver was inactive.

$11: Invalid device number (BADDNUM)

The **dev_num** parameter does not contain the device number of a device configured into the system.

$20: Invalid request code (BADREQCODE)

This error is generated only for device requests, made by SOS to a device driver, and should never be received as a result of a SOS call.

$21: Invalid status or control code (BADCTL)

The control (for D_CONTROL) or status (for D_STATUS) code is not supported by the device driver being called.

$22: Invalid control parameter list (BADCTLPARM)

The parameter list specified by the control parameter to the D_CONTROL call is not in the proper format for the control request being made.

$23: Device not open (NOTOPEN)

The character device being referenced has not been opened by the file OPEN call.

$25: Resources not available (NORESC)

The device driver is unable to acquire the system resources (such as memory, I/O ports, or interrupts) it needs to operate. This error can also occur during a file OPEN call.

$26: Call not supported on device (BADOP)

The requested SOS call is not supported by the device.

$27: I/O error (IOERR)

The device driver is unable to exchange information with the device, due to a bad storage medium or communication line, or some other cause. If this happens on a flexible disk, remove and replace the disk, and try again.

$2B: Device write-protected (NOWRITE)

The medium in this block device is write-protected. Remove the write-protect tab and try again.

$2E: Disk switched (DISKSW)

The medium in the block device has been removed and possibly replaced. This message is merely a warning, and occurs only the first time the call is made: the second time the call is made, it will be executed.

Errors $3Ø through $3F are returned by individual device drivers, and relate to specific error conditions within those drivers. The error codes generated by a device driver are described in the manual describing that device driver.

11

Memory Calls and Errors

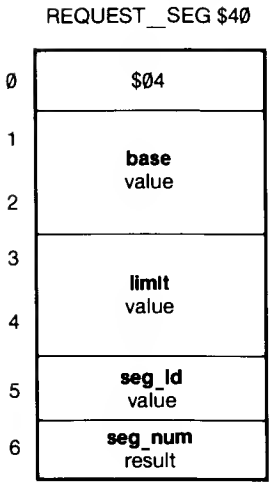
11.1 Memory Calls

These calls are used by SOS to allocate memory for interpreters, as explained in section 2.3.

$40: REQUEST_SEG
$41: FIND_SEG
$42: CHANGE_SEG
$43: GET_SEG_INFO
$44: GET_SEG_NUM
$45: RELEASE_SEG

11.1.1 REQUEST_SEG — Memory Call $40

This call requests the contiguous region of memory bounded by the **base** and **limit** segment addresses. A new segment is created if and only if no other segment currently occupies part or all of the requested region of memory.



Required Parameters

base: 2 byte value
Range: $0020..$10FF

This is the segment address (bank followed by page) of the beginning of the memory range requested.

limit: 2 byte value
Range: $0020..$10FF

This is the segment address of the end of the memory range requested.

seg_id: 1 byte value
Range: $00..$7F

This is the segment identification code of the requested segment. The caller can use this parameter to identify the type of information that the segment will contain.

The highest four bits of the **seg_id** identify the owner of the segment:

Seg_id_range	Owner	Contents
$00 to $0F	SOS Kernel	System code
$10 to $1F	Interpreter	Interpreter data
$20 to $7F	User	User program and data

The memory system does not check this parameter to ensure that it is in the proper range.

seg_num: 1 byte result
Range: $Ø1..$1F

If the requested segment is available, this parameter returns the segment number of the segment granted. This number must be used to identify the segment in subsequent calls to CHANGE_SEG, RELEASE_SEG, or GET_SEG_INFO.

Comments

Both the **base** and **limit** segment addresses must reside in switchable banks $ØØ through $ØE, system bank $ØF, or system bank $1Ø. In addition, the **base** address must be less than or equal to the **limit** address. If the **base** and **limit** segment address parameters fail to meet the above criteria, then the segment will not be allocated and error BADBKPG will be returned.

The ranges for **base** and **limit** are not continuous: these are the allowable segment addresses:

$ØØ2Ø..$ØØ9F
$Ø12Ø..$Ø19F
.
.
$ØE2Ø..$ØE9F
$ØFØØ..$ØF1F
$1ØAØ..$1ØFF

v1.2 SOS can keep track of $1F segments

Errors

$EØ:	BADBKPG	Invalid segment address (bank/page pair)
$E1:	SEGRQDN	Segment request denied
$E2:	SEGTBLFULL	Segment table full

11.1.2 FIND_SEG

Memory Call $41

This call searches memory from high memory down, until it finds the first free space that is **pages** pages long and meets the search restrictions in **search_mode**. If such a space is found, it assigns this free space to the caller as a segment (as in REQUEST_SEG), returning both the segment number and the location in memory of the segment. If a segment with the specified size is not found, then the size of the largest free segment which meets the given criterion will be returned in **pages**. In this case, however, error SEGRQDN will be returned, indicating that the segment was not created.

FIND_SEG $41

Ø	$Ø6
1	**search_mode** value
2	**seg_id** value
3	**pages** value/result
4	
5	**base** result
6	
7	**limit** result
8	
9	**seg_num** result

Required Parameters

search_mode: 1 byte value
Range: $ØØ..$Ø2

This parameter selects one of three constraints to place upon the segment search:

$ØØ: may not cross a 32K bank boundary
$Ø1: may cross one 32K bank boundary
$Ø2: may cross any 32K bank boundary

seg_id: 1 byte value
Range: $00..$7F

This is the segment identification code of the requested segment. The caller can use this parameter to identify the type of information that the segment will contain.

The highest four bits of the **seg_id** identify the owner of the segment:

Seg_id_range	Owner	Contents
$00 to $0F	SOS Kernel	System code
$10 to $1F	Interpreter	Interpreter data
$20 to $7F	User	User program and data

The memory system does not check this parameter to ensure that it is in the proper range.

pages: 2 byte value/result
Range: $0001..$FFFF

This is the the number of contiguous pages to search for. If no free space is found that contains this many pages, then the memory system will return in this parameter the size of the largest free space it can find; the SEGRQDN error is also generated. A page count of $00 always returns error BADPGCNT.

base: 2 byte result
Range: $0020..$0E9F

This is the the segment address of the beginning of the new segment.

limit: 2 byte result
Range: $0020..$0E9F

This is the segment address of the end of the new segment.

seg_num: 1 byte result
Range: $01..$1F

This is the the segment number of the segment granted. This number must be used to identify the segment in subsequent calls to CHANGE_SEG, RELEASE_SEG, or GET_SEG_INFO.

Comments

FIND_SEG does not search the system banks $0F and $10.

The **base** and **limit** parameters both return $0000 if the segment is not granted; even though **pages** returns the length of the largest available segment, **base** and **limit** do not return its location.

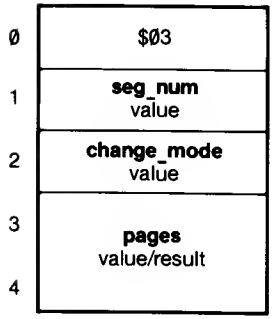
Errors

$E1:	SEGRQDN	Segment request denied
$E2:	SEGTBLFULL	Segment table full
$E5:	BADSRCHMODE	Invalid search mode parameter
$E7:	BADPGCNT	Invalid pages parameter ($00)

11.1.3 CHANGE_SEG Memory Call $42

This call changes either the **base** or **limit** segment address of the specified segment by adding or releasing the number of pages specified by the **pages** parameter. If the requested boundary change overlaps an adjacent segment or the end of the memory, then the change request is denied, error SEGRQDN is returned, and the maximum allowable page count is returned in the **pages** parameter.

CHANGE_SEG $42



Required Parameters

seg_num: 1 byte value
Range: $01..$1F

This is the segment number of the segment to be changed.

change_mode: 1 byte value
Range: $00..$03

The change mode indicates which end (**base** or **limit**) of the segment to change, and whether to add or release space at that end.

$00: Release from the **base** (decrease size)
$01: Add before the **base** (increase size)
$02: Add after the **limit** (increase size)
$03: Release from the **limit** (decrease size)

pages: 2 byte value/result
Range: $0001..$FFFF

This is the number of pages to add to or release from the segment. If too many pages are added to or removed from the segment, then the segment is not changed, and the maximum number of pages that can be added or removed in the requested **change_mode** is returned in this parameter, along with a SEGRQDN error.

Comments

You cannot move both ends of a segment at once.

If the segment was granted by FIND_SEG, a CHANGE_SEG operation will not heed the bank-crossing criterion that was used in finding the segment. If you request a segment that does not cross a bank boundary, then increase it with CHANGE_SEG, the larger segment may cross a bank boundary.

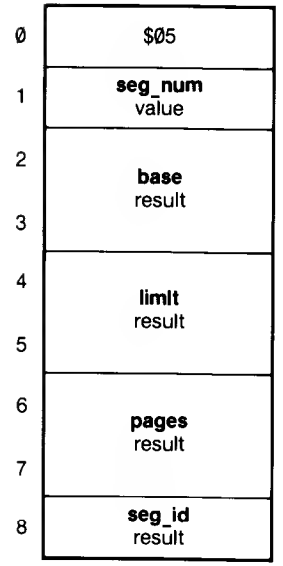
Errors

$E1	SEGRQDN	Segment request denied
$E3	BADSEGNUM	Invalid segment number
$E6	BADCHGMODE	Invalid change mode parameter

11.1.4 GET_SEG_INFO Memory Call $43

This call returns the beginning and ending locations, size in pages, and identification code of the segment specified by **seg_num**.

GET_SEG_INFO $43



Required Parameters

seg_num: 1 byte value
Range: $01..$1F

This returns the segment number of an existing segment.

base: 2 byte result
Range: $0020..$109F

This returns the segment address of the beginning of that segment.

limit: 2 byte result
Range: $0020..$109F

This returns the segment address of the end of that segment.

pages: 2 byte result
Range: $0001..$FFFF

This returns the number of pages contained by the segment.

seg_id: 1 byte result
Range: $00..$7F

This returns the identification code of the segment. The highest four bits of the **seg_id** identify the owner of the segment:

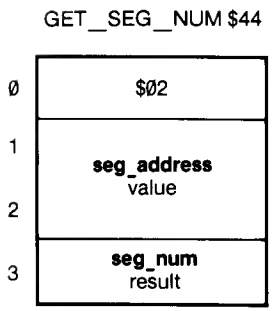
Seg_id_range	Owner	Contents
$00 to $0F	SOS Kernel	System code
$10 to $1F	Interpreter	Interpreter data
$20 to $7F	User	User program and data

Errors

$E3:	BADSEGNUM	Invalid segment number

11.1.5 GET_SEG_NUM Memory Call $44

This call returns the segment number of the segment, if any, that contains the specified segment address.



Required Parameters

seg_address: 2 byte value
Range: $0020..$109F

This is the segment address in question.

seg_num: 1 byte result
Range: $01..$1F

This is the segment number of the segment that contains the specified segment address.

Comments

You may make a subsequent call to GET_SEG_INFO with the resultant segment number to determine the ownership of that segment.

Errors

$E0:	BADBKPG	Invalid segment address (bank/page pair)
$E4:	SEGNOTFND	Segment not found

11.1.6 RELEASE_SEG Memory Call $45

RELEASE_SEG $45

0	$01
1	**seg_num** value

This call releases the memory occupied by the segment specified by **seg_num**, by removing the segment from the segment table. The space formerly occupied by the released segment is returned to free memory. If **seg_num** equals zero, then all nonsystem segments (those with segment identification codes greater than $0F) will be released.

Required Parameters

seg_num: 1 byte value
Range: $00..$1F

This is the segment number of the segment to be released. If **seg_num** is $00, then all segments not owned by SOS are released.

Errors

$E3	BADSEGNUM	Invalid segment number

11.2 Memory Call Errors

The errors below are generated by SOS memory calls. For other errors, see Appendix D.

$E0: Invalid segment address (BADBKPG)

The segment address has an invalid bank number, page number, or both.

$E1: Segment request denied (SEGRQDN)

No segment can be created that meets the caller's size and boundary criteria.

$E2: Segment table full (SEGTBLFULL)

SOS can keep track of no more segments: existing segments must be released or consolidated if more segments are needed.

SOS can keep track of $1F segments.

$E3: Invalid segment number (BADSEGNUM)

The **seg_num** passed is not that of a currently existing segment.

$E4: Segment not found (SEGNOTFND)

For GET_SEG_NUM, no segment in the system contains the segment address specified.

$E5: Invalid search_mode parameter (BADSRCHMODE)

For FIND_SEG, the **search_mode** parameter is invalid (greater than $02).

$E6: Invalid change_mode parameter (BADCHGMODE)

For CHANGE_SEG, the **change_mode** parameter is invalid (greater than $03).

$E7: Invalid pages parameter (BADPGCNT)

The **pages** parameter is invalid (equal to $00).

12

Utility Calls and Errors

12.1 Utility Calls

The following system calls deal with the system clock/calendar, the event fence, the analog input ports, and other general system resources.

$60: SET_FENCE
$61: GET_FENCE
$62: SET_TIME
$63: GET_TIME
$64: GET_ANALOG
$65: TERMINATE

12.1.1 SET_FENCE — Utility Call $60

This call changes the current value of the user event fence to the value specified in the **fence** parameter.

SET_FENCE $60

0	$01
1	**fence** value

Required Parameters

fence: 1 byte value
Range: $00..$FF

This parameter contains the new value of the user event fence for the operating system's event mechanism. Events with priority less than or equal to the fence will not be serviced until the fence is lowered.

Errors

No errors are possible.

12.1.2 GET_FENCE — Utility Call $61

This call returns the current value of the user event fence.

GET_FENCE $61

Ø	$Ø1
1	**fence** result

Required Parameters

fence: 1 byte result
Range: $ØØ..$FF

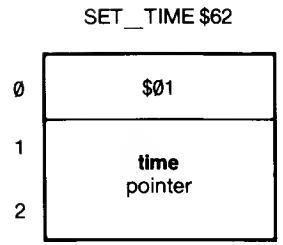
This parameter returns the current setting of the user event fence. Events with priority less than or equal to the fence will not be serviced until the fence is lowered.

Errors

No errors are possible.

12.1.3 SET_TIME — Utility Call $62

This call sets the system clock to the contents of a buffer located at the specified address. If the system has no functioning clock, SET_TIME stores the contents of the buffer as the last valid time, to be returned on the next GET_TIME call.



Required Parameters

time: pointer

This is a pointer to an 18-byte buffer containing the current date and time. The information is specified as an 18-byte ASCII string whose format is

Y Y Y Y M M D D X H H N N S S X X X

The meaning of each field is as below:

Field	Meaning	Minimum	Maximum
YYYY:	Year	19ØØ	1999
MM:	Month	ØØ or Ø1	12 (December)
DD:	Date	ØØ or Ø1	28, 3Ø, or 31
X:	Ignored		
HH:	Hour	ØØ(Midnight)	23 (11:ØØ p.m.)
NN:	Minute	ØØ	59
SS:	Second	ØØ	59
XXX:	Ignored		

For example, December 29, 198Ø, at 9:3Ø a.m., would be specified by the string "198Ø122900930000ØØ".

Comments

On input, SOS replaces the first two digits of the year with "19" and ignores the day of the week and the millisecond. SOS calculates the day from the year, month, and date.

SOS does not check the the validity of the input data to make sure each field is in the proper range. The clock makes several restrictions: it rejects any invalid combination of month and date. The clock only accepts dates in the range 1..30 if the month is 4, 6, 9, or 11; it only accepts dates in the range 1..28 if the month is 2: February 29 is always rejected.

SET_TIME attempts to set the hardware clock, whether or not it is present and functioning. It also stores the new time in system RAM as the last known valid time; this time will be returned by all subsequent GET_TIME calls if the hardware clock is missing or malfunctioning.

The clock does not roll over the year.

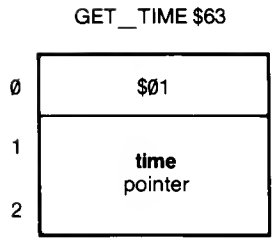
The format of the SET_TIME string is the same as that of the GET_TIME result, except that SET_TIME ignores the day of the week and the millisecond fields.

Errors

No errors are possible.

12.1.4 GET_TIME Utility Call $63

This call reads the time from the system clock and returns it to the buffer located at the specified address. If the system has no functioning clock, GET_TIME returns the last known valid time.



Required Parameters

time: pointer

This is a pointer to an 18-byte buffer containing the current date and time. The information is specified as an 18-byte ASCII string whose format is

Y Y Y Y M M D D W H H N N S S U U U

The meaning of each field is as below:

Field	Meaning	Minimum	Maximum
YYYY:	Year	1900	1999
MM:	Month	00 or 01	12 (December)
DD:	Date	00 or 01	28, 30, or 31
W:	Day	01 (Sunday)	07 (Saturday)
HH:	Hour	00 (Midnight)	23 (11:00 p.m.)
NN:	Minute	00	59
SS:	Second	00	59
UUU:	Millisecond	000	999

For example, Friday, March 21, 1980, at 1:27:41.001 p.m., would be returned as "19800321613274100 1".

Comments

If the hardware clock is not operational, the utility manager retrieves the last known valid time from system RAM. If no last known valid time is stored, GET_TIME returns a string of eighteen ASCII zeros: "000000000000000000".

SOS calculates the day of the week from the year, month, and date.

The clock will only generate dates in the range 1..3Ø if the month is 4, 6, 9, or 11; it will only generate dates in the range 1..28 if month is 2: February 29 will never be generated by a system with a functioning clock. A system without a functioning clock can return February 29 if that month and date have been set by a SET_TIME call.

The clock does not roll over the year.

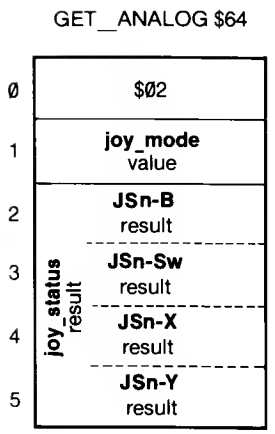
You must ensure that the buffer pointed to by **time** can hold all eighteen ($12) bytes, to avoid overwriting other data.

Errors

No errors are possible.

12.1.5 GET_ANALOG — Utility Call $64

This call reads the analog and digital inputs from an Apple III Joystick connected to port A or B on the back of the Apple III.



Required Parameters

joy_mode: 1 byte value
Range: $ØØ..$Ø7

This parameter specifies the joystick inputs to be read. For each value of **joy_mode**, the following inputs will be read:

Joy_mode	Port	Buttons/Switches	Horizontal	Vertical
$ØØ	B	JSØ-B, JSØ-Sw	—	—
$Ø1	B	JSØ-B, JSØ-Sw	JSØ-X	—
$Ø2	B	JSØ-B, JSØ-Sw	—	JSØ-Y
$Ø3	B	JSØ-B, JSØ-Sw	JSØ-X	JSØ-Y
$Ø4	A	JS1-B, JS1-Sw	—	—
$Ø5	A	JS1-B, JS1-Sw	JS1-X	—
$Ø6	A	JS1-B, JS1-Sw	—	JS1-Y
$Ø7	A	JS1-B, JS1-Sw	JS1-X	JS1-Y

The names for these variables are those used in the *Apple III Owner's Guide*, Appendix C. These eight variables are returned by the **joy_status** parameter.

joy_status: 4 byte result
Range: $00000000..$FFFFFFFF

This 4-byte field is treated as one parameter by SOS. Here we subdivide it into four 1-byte fields for clarity; *n* represents the numbers of the joystick (1 or 2) as determined by the **joy_mode** parameter.

JSn-B: 1 byte result
Range: $00..$FF

This digital output returns $00 if the button is off and returns $FF if the button is on.

JSn-Sw: 1 byte result
Range: $00..$FF

This digital output returns $00 if the switch is off and returns $FF if the switch is on.

JSn-X: 1 byte result
Range: $00..$FF

This analog output returns a value from $00 to $FF corresponding to the horizontal position of the joystick. A position that was not read (due to the **joy_mode** parameter) returns a byte of $00.

JSn-Y: 1 byte result
Range: $00..$FF

This analog output returns a value from $00 to $FF corresponding to the vertical position of the joystick. A position that was not read (due to the **joy_mode** parameter) returns a byte of $00.

Comments

An input device other than a joystick can be read, provided (a) it uses the same pins for analog and digital inputs, and (b) each pin produces the correct signals, as described in the *Apple III Owner's Guide*.

Both buttons of the selected joystick are always read and returned.

Reading the analog inputs slows down the execution speed of this call and should be avoided when unnecessary.

JSn-B, **JSn-Sw**, **JSn-X**, and **JSn-Y** all return results of $FF if no joystick is attached to the port.

The XNORESRC error will be generated if an attempt is made to read Port A and a device driver (such as the Silentype driver) has already claimed the use of that port.

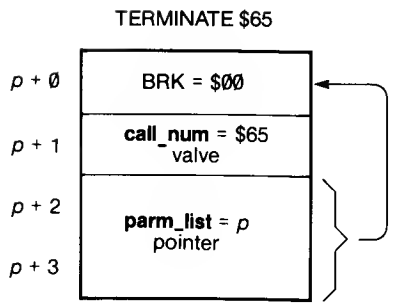
The **parm_count** is $02, *not* $05.

Errors

$25	XNORESRC	Resource not available
$70	BADJMODE	Invalid joystick mode

12.1.6 TERMINATE — Utility Call $65

This call clears memory, clears the screen, and displays INSERT SYSTEM DISKETTE & REBOOT in 40-column black-and-white text mode on the screen. The system then hangs, and waits for the user to press CONTROL-RESET and reboot.



Required Parameters

None

Comments

Only the SOS Call Block is shown for this call. Since this call has no parameters, the **parameter_count** is $00. Thus the **parameter_list** pointer must point to a byte containing $00. The most convenient such byte is the BRK opcode beginning the TERMINATE call, so this call customarily bites its own tail.

Before issuing a TERMINATE call, the interpreter should close all open files. This will ensure that all I/O buffers are written out, and all file entries updated, while the necessary information still exists.

This call is the recommended way to leave a program. It provides a clean exit to a program, and leaves no traces of it in memory for the user's examination. It can be used in conjunction with a copy-protection scheme to protect a program from piracy. It also provides a hook that could be used to return control to a future command interpreter.

Errors

No errors are possible. This is an excellent call for beginners.

12.2 Utility Call Errors

One error can be generated by one of the utility calls; other errors are listed in Appendix D.

$70: Invalid Joystick Mode (BADJMODE)

The **joy_mode** parameter is greater than $07.

A

SOS Specifications

Version: SOS 1.1, 1.2 and 1.3

Classification:

Single-task, configurable, interrupt-driven operating system.

File system—hierarchical, tree file structure.

Device-Independent I/O.

CPU Architecture:

Address enhanced 65Ø2 instruction set.

Supports both bank-switched and enhanced indirect addressing.

Separate execution environments for user and SOS including private zero and stack pages.

System Calls:

Based on 65Ø2 BRK instruction, pointer, and value parameter types.

Error codes returned via A register.

All other CPU registers preserved upon return.

Optional parameter lists for future expansion.

File Management System:

Hierarchical file structure.

Pathname prefix facility.

Byte-oriented file access to both directory/user files and device files.

Dynamic, non-contiguous file allocation on block devices.

Automatic buffering (current index block and data block).

Dynamic memory allocation of file buffers.

Block size (512 bytes).

File protection: rename/destroy/read/write access attributes.

File level assignment on Open.

Automatic date/time stamping of files.

Automatic volume logging/swapping, supported by system message center.

Multiple volumes per block device can be "open" simultaneously.

Sparse file capability:

maximum number of active volumes = 8

maximum disk size = 32 Mbytes

maximum user file size = 16 Mbytes

maximum file entries in volume directory = 51

maximum file entries in a subdirectory = 1663

file names — maximum 15 characters

pathnames — maximum 128 characters

File system calls:

CREATE	READ
DESTROY	WRITE
RENAME	CLOSE
SET_FILE_INFO	FLUSH
GET_FILE_INFO	SET_MARK
VOLUME	GET_MARK
SET_PREFIX	SET_EOF
GET_PREFIX	GET_EOF
OPEN	SET_LEVEL
NEWLINE	GET_LEVEL

Device Management System:

Block and character device classes.

Standardized interface for block and for character devices.

All devices are named and configurable.

Support for synchronous, interrupt, and DMA-based I/O.

maximum number of devices = 24

maximum number of block devices = 12

Device system calls:

GET_DEV_NUM	D_STATUS
D_INFO	D_CONTROL

Memory/Buffer Management System:

All memory allocated as segments.

Supports maximum of 512 Kbytes RAM.

System buffers allocated and released dynamically.

System buffer checksum routine for data integrity.

Memory system calls:

REQUEST_SEG	GET_SEG_INFO
FIND_SEG	GET_SEG_NUM
CHANGE_SEG	REL_SEG

Additional System Functions:

System clock/calendar
(year/month/day/weekday/hour/minute/second/ms).

Joysticks: reads X and Y axes, pushbutton, and switch.

TERMINATE call provides clean program termination and clears memory.

System calls:

SET_TIME	TERMINATE
GET_TIME	GET_ANALOG

Interrupt Management System:

Receives hardware interrupts (IRQ, NMI) and system calls (BRK).

Hardware resource allocation and deallocation.

Dispatches to driver interrupt handlers.

Event Management System:

Priority-based event signaling.

Event handlers preempted by higher priority events.

Events with equal priorities process FIFO.

Event fence delays events with priority less than fence.

Event system calls:

SET_FENCE	GET_FENCE

System Configuration:

Menu-driven system-configuration editor (System Configuration Program).

Can add, remove, and modify drivers and can select the keyboard-layout and system-character-set tables.

Standard Device Drivers:

Floppy disk (.D1, .D2, .D3, .D4)

143,360 bytes (formatted) per volume.

Automatically reports mounting of a new volume.

Built into SOS kernel.

Console (.CONSOLE)

Interrupt-driven keyboard (supports type-ahead).

Configurable keyboard-layout table (via SCP).

Raw-keystroke and no-wait input modes.

Event handler supports anykey and attention character.

Optional screen echoing.

Console control modes:

video on/off
flush type-ahead buffer
suspend screen output
display control characters
flush screen output

Cursor positioning commands.

Viewport set, clear, save, and restore commands.

Horizontal and vertical scrolling.

Text modes: 24 × 8Ø and 24 × 4Ø B&W and 24 × 4Ø color (normal and inverse).

Configurable system character set table (via SCP).

Character set can be changed under program control at any time.

Screen read command.

Graphics (.GRAFIX)

Displays graphical and textual information simultaneously.

Graphics modes:
56Ø × 192 and 28Ø × 192 in B&W video.
28Ø × 192 and 14Ø × 192 in 16 colors.

Point-plotting and line-drawing commands using graphics viewport and pen.

Raster block picture operations.

Color operator table, controls color overwrite.

Transfer modes allow binary operations on the drawing color and the current screen color.

Allows use of either the system character set or an alternate character set to display ASCII text on the screen.

Single or dual graphics screens.

General purpose communications (.RS232)

RS-232-C interface.

Configurable data rates from 11Ø to 96ØØ baud.

Configurable protocols, including XON/XOFF, ETX/ACK, and ENQ/ACK.

Interrupt-driven, buffered, bi-directional data transfer.

Hardware handshaking option.

Serial printer (.PRINTER)

RS-232-C interface.

Configurable data rates from 11Ø to 96ØØ baud.

Interrupt-driven and buffered (output only).

Hardware handshaking option.

Audio (.AUDIO)

64 volume levels.

Produces tones from 31 to 5Ø9Ø Hz (over 7 octaves).

Duration range from Ø to 5 sec (increments of 1/6Ø sec).

B

ExerSOS

ExerSOS is a interactive BASIC program that lets you make SOS calls from the keyboard without writing a special assembly-language program to test each call. It is intended to let you try out calls to see how they work. ExerSOS lets you choose a call from a menu, then prompts you for each of the call's input parameters, and gives you the correct output parameters or error message.

B.1 Using ExerSOS

To use ExerSOS, insert the ExerSOS disk into the built-in drive and press CONTROL-RESET. After the introductory displays you will see the Main Menu.

B.1.1 Choosing Calls and Other Functions

The Main Menu presents you with a choice of functions. Typing Ø will EXIT ExerSOS. The first 35 of these functions are SOS calls (listed below by type). The remainder are special functions available within ExerSOS. The full list of functions is

File Calls:

$C0: CREATE
$C1: DESTROY
$C2: RENAME
$C3: SET_FILE_INFO
$C4: GET_FILE_INFO
$C5: VOLUME
$C6: SET_PREFIX
$C7: GET_PREFIX
$C8: OPEN
$C9: NEWLINE
$CA: READ
$CB: WRITE
$CC: CLOSE
$CD: FLUSH
$CE: SET_MARK
$CF: GET_MARK
$DØ: SET_EOF
$D1: GET_EOF
$D2: SET_LEVEL
$D3: GET_LEVEL

Device Calls:

$82: D_STATUS
$83: D_CONTROL
$84: GET_DEV_NUM
$85: D_INFO

Memory Calls:

$4Ø: REQUEST_SEG
$41: FIND_SEG
$42: CHANGE_SEG
$43: GET_SEG_INFO
$44: GET_SEG_NUM
$45: RELEASE_SEG

Utility Calls:

$6Ø: SET_FENCE
$61: GET_FENCE
$62: SET_TIME
$63: GET_TIME
$64: GET_ANALOG

ExerSOS Utilities:

$1: Display Directory
$2: Display Open Files
$3: Display Active Memory Segments
$4: Display/Edit Contents of Data Buffer

B.1.2 Input Parameters

When you select a SOS call from the Main Menu, the display is replaced by a split-screen menu showing the name of the call at the top. The left half of the screen is used for typing input parameters to the call; the right is used to show the resultant SOS call error and any output parameters. You will then be prompted for each input parameter, following the description of the call in the SOS Manual. If you wish to return to the Main Menu, type a backslash (\) and press RETURN.

All parameters have the same names as in this manual, and appear in the same order as in the description of the SOS call in Volume 2. Pointer parameters, however, are omitted, as all values and results are passed interactively, rather than by building a table in memory and passing its address.

In some cases, a range of legal values is displayed; if your entry falls outside that range, you will be prompted again. For example, the first prompt you encounter in the READ call is

```
ref_num [0..255] -
```

If you respond to this with an out-of-range value, the prompt will be repeated.

You may also type data in hexadecimal by preceeding a value with a dollar sign ($). Some input fields have a fixed dollar sign: these fields require hex input. SOS calls requiring no input display

```
[None]
```

before reporting the results of the call.

When typing an input parameter, you can use the ESCAPE key to edit the input, as in BASIC.

Several SOS calls employ an optional parameter list along with a **length** parameter. For those calls, ExerSOS asks you for the **length** and selectively prompts or displays information as requested.

B.2 The Data Buffer

ExerSOS maintains two buffers you should be aware of: the data buffer and the string buffer. ExerSOS alone locates the 16K data buffer in memory. All I/O operations (READ, WRITE) use the data buffer. Hence, a READ call followed by a WRITE call will transfer bytes from one file to another.

In order to ensure the return of this 16K space to the system, always exit ExerSOS through the Main Menu, never by typing CONTROL-C. If you should accidentally exit ExerSOS, reboot by pressing CONTROL-RESET.

B.2.1 Editing the Data Buffer

The Display/Edit function allows you to select any of the 64 256-byte pages of memory occupied by the data buffer, and displays that page in hex with the ASCII equivalents on the right side of the screen. You are then placed in Edit mode with the cursor (denoted by matching "[..]") positioned in the upper-right corner. You can move the cursor through the use of the four arrow keys.

You can alter the contents of a byte by typing a hex digit, (that is, 0..9, A..F, a..f). Note that as you do so, the value you type is placed in the low-order nibble of the target byte, and the value that was in the low-order nibble moves to the high-order nibble. You may terminate the input to a byte by pressing RETURN, which accepts the new value, or ESCAPE, which restores the original value.

If you press ESCAPE while you are in the cursor-positioning phase, you exit from Edit mode and have the choice of returning to the Main Menu or displaying another page of the buffer.

B.3 The String Buffer

The string buffer is used by many of the calls as temporary storage any time a pathname or device name is passed into or out of a SOS call. Additionally, the D_STATUS and D_CONTROL calls use the string buffer for the STATUS_LIST and CONTROL_LIST, respectively.

The following SOS calls require some further user input:

D_STATUS

In addition to the SOS-required input parameters, ExerSOS prompts you for two more items. The first prompt,

```
Initialize Buffer [Y/N] —
```

lets you initialize the string buffer by typing Y, or leave its current contents intact by typing N. Usually, you will initialize it, to make sure no garbage from a previous call obscures your results. However, in some cases, you may wish to make a status call, then change something with a control call, then check the buffer with a status call again: in such a case do not initialize the buffer.

The second prompt,

```
Amount of output —
```

asks you how many bytes of the string buffer you wish to see. If you specify more bytes than are in the status list, the remaining bytes will be either zeros or garbage, depending on your response to the "Initialize?" prompt.

D_CONTROL

After you specify the **dev_num** and **control_code**, ExerSOS allows you to specify the control list from either of two places. If you type a "Ø" to the "Length of input" prompt, the call is made from the current value of the string buffer. If you respond to the prompt with a value larger than Ø, you are prompted for each byte of the control list. The resultant string is moved into the string buffer.

B.4 Leaving ExerSOS

To leave ExerSOS, return to the Main Menu and type Ø. You will be asked to confirm your intention: type Y to exit (any other reply will return you to the Main Menu). ExerSOS will drop into BASIC, and you will be able to run another BASIC program, or reboot by pressing CONTROL-RESET. If you leave ExerSOS inadvertently, as by typing CONTROL-C, you should reboot. If you try to RUN the program without rebooting, you will have lost the 16K space allocated to the data buffer.

MakeInterp

MakeInterp is a program that takes an assembly-language code file produced by the Apple III Pascal Assembler and converts it to the proper format for a bootable SOS.INTERP file. If you are writing an interpreter, this makes it unnecessary for you to know the details of interpreter file format, and protects you from future changes in this format.

To use MakeInterp, boot Pascal and insert the ExerSOS disk into, say, .D2. Now execute (that is, type X)

```
.D2/MAKEINTERP.CODE
```

Then type the input pathname, the name of the interpreter code file, for example,

```
.D2/INTERP.CODE
```

and the output pathname, say,

```
.D2/SOS.INTERP
```

As the disk spins, you see this message displayed:

```
Converting Files
```

When the conversion is complete, MakeInterp displays the message

```
Files converted
```

and returns you to the Pascal command line.

All pathnames must be complete, with suffix. If you type any invalid input, you will have to execute the program again.

D

Error Messages

SOS detects two types of errors:

- Non-fatal SOS errors, occurring during a SOS call, that are detected and flagged;
- Fatal SOS errors, occurring during a SOS call or interrupt sequence, that signal such a substantial irregularity that the system cannot continue to operate.

In addition, the SOS bootstrap loader detects bootstrap errors, which occur only when the system is starting up.

The reporting mechanism for non-fatal SOS errors is discussed in Volume 1, section 8.4. The error code is returned in the accumulator after a SOS call: an error code of $ØØ means no error was encountered in the call. The error code is normally used by the interpreter to display a message to the user, to repeat an operation, or to take some other action.

Bootstrap errors and fatal errors occur when an error condition is so critical that no recovery is possible. These errors cause their own messages to be displayed on the screen, as no interpreter is in place to interpret them. These errors are discussed in detail in section D.3.

D.1 Non-Fatal SOS Errors

Explanations of the general system errors are given in section 8.4 of Volume 1. Explanations of the other non-fatal system errors are given in Volume 2. The list below, numerically ordered, is for easy reference. Three things are listed for each error: the error number, a suggested name for the assembly-language routine handling the error, and a suggested error message for the interpreter to display on the screen.

D.1.1 General SOS Errors
(See section 8.4)

$Ø1: BADSCNUM	Invalid SOS call number
$Ø2: BADCZPAGE	Invalid caller zero page
$Ø3: BADXBYTE	Invalid indirect pointer X-byte
$Ø4: BADSCPCNT	Invalid SOS call parameter count
$Ø5: BADSCBNDS	SOS call pointer out of bounds

D.1.2 Device Call Errors
(See section 1Ø.2)

$1Ø: DNFERR	Device not found
$11: BADDNUM	Invalid device number
$2Ø: BADREQCODE	Invalid request code
$21: BADCTLCODE	Invalid status or control code
$22: BADCTLPARM	Invalid control parameter list
$23: NOTOPEN	Device not open
$25: NORESRC	Resources not available
$26: BADOP	Invalid operation
$27: IOERROR	I/O error
$2B: NOWRITE	Device write-protected
$2E: DISKSW	Disk switched
$3Ø..$3F:	Device-specific errors

D.1.3 File Call Errors
(See section 9.2)

$4Ø: BADPATH	Invalid pathname syntax
$41: CFCBFULL	Character File Control Block full
$42: FCBFULL	Block File or Volume Control Block full
$43: BADREFNUM	Invalid file reference number
$44: PATHNOTFND	Path not found
$45: VNFERR	Volume not found
$46: FNFERR	File not found
$47: DUPERR	Duplicate file name
$48: OVRERR	Overrun on volume
$49: DIRFULL	Directory full
$4A: CPTERR	Incompatible file format
$4B: TYPERR	Unsupported storage type
$4C: EOFERR	End of file would be exceeded
$4D: POSNERR	Position out of range
$4E: ACCSERR	Access not allowed
$4F: BTSERR	Buffer too small
$5Ø: FILBUSY	File busy
$51: DIRERR	Directory error
$52: NOTSOS	Not a SOS volume
$53: BADLSTCNT	Length parameter invalid
$55: BUFTBLFULL	Buffer table full

$56: BADSYSBUF	Invalid system buffer parameter
$57: DUPVOL	Duplicate volume
$58: NOTBLKDEV	Not a block device
$59: LVLERR	Invalid level
$5A: BITMAPADDR	Invalid bit map address

D.1.4 Utility Call Errors
(See section 12.2)

$70: BADJOYMODE	Invalid joy_mode parameter

D.1.5 Memory Call Errors
(See section 10.2)

$E0: BADBKPG	Invalid segment address
$E1: SEGRQDN	Segment request denied
$E2: SEGTBLFULL	Segment table full
$E3: BADSEGNUM	Invalid segment number
$E4: SEGNOTFND	Segment not found
$E5: BADSRCHMODE	Invalid search_mode parameter
$E6: BADCHGMODE	Invalid change_mode parameter
$E7: BADPGCOUNT	Invalid pages parameter

D.2 Fatal SOS Errors

If SOS encounters an internal error from which it cannot recover, it displays an error message (including the code number of the error that occurred) on the screen, beeps the speaker, and hangs. The only recovery possible is to reboot.

The fatal error codes and conditions are listed below. The phrase following the number is a convenient name for the error, but no interpreter will be able to display it to the user, as SOS will not be around to help.

$01: Invalid BRK (BADBRK)

A BRK software interrupt was encountered within SOS. As SOS is not reentrant, it is not allowed to make SOS calls to itself; making such a call is an unrecoverable error and means that the memory region containing SOS has been scrambled.

$02: Invalid interrupt (BADINT)

An interrupt occurred that cannot be acknowledged by SOS. The 6502's IRQ or NMI line was pulled down, but either polling did not reveal the device that performed the interrupt, or no device driver had claimed that interrupt.

$04: Invalid NMI (NMIHANG)

A request was made for SOS to lock the RESET/NMI key, but a device is currently attempting to perform a NMI interrupt. If the interrupt is not granted and handled within a short time after the request to lock NMI was made, this error will occur.

$05: Event queue overflow (EVQOVFL)

More events (see Chapter 6) have occurred than have been handled. Possibly the event fence is set too high, and few events are being handled.

$06: SOS stack overflow (STKOVFL)

The SOS stack has been pushed to more than 256 bytes, and the data at the bottom of the stack have been overwritten.

$07: Invalid control or status request (BADSYSCALL)

The device system has detected an invalid control or status request.

$08: Too many drivers (MCTOVFL)

Too many device drivers have been created for SOS to keep track of.

$09: Memory too small (MEM2SML)

The Apple III's memory is too small for SOS to operate in; that is, less than 128K bytes.

$0A: Buffer Control Block damaged (VCBERR)

The file system's Buffer Control Block has been damaged due to a memory failure.

$0B: File Control Block damaged (FCBERR)

The file system's File Control Block has been damaged due to a memory failure.

$0C: Invalid allocation blocks (ALCERR)

Allocation blocks are invalid.

$0E: Pathname too long (TOOLONG)

A pathname supplied or internally generated contains more than 256 characters. This can result from concatenating a long prefix to a long filename.

$0F: Invalid buffer number (BADBUFNUM)

An internal buffer allocation request has supplied an invalid buffer number.

$10: Invalid buffer size (BADBUFSIZ)

An internal buffer allocation request has supplied an invalid buffer size.

D.3 Bootstrap Errors

If an error occurs during the bootstrap operation, an error message is displayed (in uppercase inverse characters) in the middle of the video screen, the speaker beeps, and the system hangs. Bootstrap errors are not SOS errors, as they occur before SOS has started running: for this reason, they are not numbered. Any bootstrap error is a fatal error: you must insert a proper boot diskette, then hold down the CONTROL key and press the RESET button to reboot.

The following errors can be produced during a bootstrap operation:

DRIVER FILE NOT FOUND

There is no file named SOS.DRIVER listed in the volume directory of the boot disk. SOS cannot operate without device drivers, and the drivers must be stored in a file with this name in the volume directory of the disk.

DRIVER FILE TOO LARGE

The SOS.DRIVER file is too large to fit into the system's memory along with the interpreter. Use the System Configuration Program to remove some drivers from this file.

EMPTY DRIVER FILE

The SOS.DRIVER file contains no device drivers. SOS requires at least one device driver, .CONSOLE, to operate.

INCOMPATIBLE INTERPRETER

The interpreter is either too large or specifies a loading location that conflicts with SOS. This error usually occurs when trying to load an older interpreter with a newer version of SOS.

INTERPRETER FILE NOT FOUND

There is no file named SOS.INTERP listed in the volume directory of the boot disk. SOS cannot operate without an interpreter, and the interpreter must be stored in a file with this name, in the volume directory of the disk.

INVALID DRIVER FILE

The SOS.DRIVER file is not in the proper format for a driver file. Make sure that the file was created by the System Configuration Program or obtained from a valid Apple III boot disk.

I/O ERROR

The loader encountered an I/O error while trying to read the kernel, interpreter, or driver file from the disk in the Apple III's internal disk drive. Make sure the correct disk is properly inserted in that drive.

KERNEL FILE NOT FOUND

No file named SOS.KERNEL is listed in the volume directory of the boot disk. The files SOS.KERNEL, SOS.INTERP, and SOS.DRIVER must all be present in the volume directory of a disk to be booted.

ROM ERROR: PLEASE NOTIFY YOUR DEALER

Your Apple III contains an older version of the bootstrap ROM that is not supported by this version of SOS. Your Apple dealer should be able to replace the ROM at no cost. If you receive this message, please contact your dealer or nearest Apple Service Center.

TOO MANY BLOCK DEVICES

The SOS.DRIVER file contains too many device drivers for block devices. Use the System Configuration Program to remove some of the block device drivers from this file.

TOO MANY DEVICES

The SOS.DRIVER file, while small enough to fit into memory, contains too many device drivers for SOS to keep track of. Use the System Configuration Program to remove some drivers from this file.

E

Data Formats of Assembly-Language Code Files

Interpreters can load additional code modules. When you write an interpreter, you may want to make these code modules relocatable. This appendix describes the relocation information generated by the Apple III Pascal Assembler.

Appendix E is derived from the *Apple III Pascal Technical Reference Manual*. Read that manual if you want more detailed information.

Most of the information about assembly-language code files described in the *Apple III Pascal Technical Reference Manual* is addressed to Pascal programmers. However, if you want to use Pascal Assembler code files when you write an interpreter, you need to deal with only two general areas: the overall organization of the code file, and the data structures generated for various psuedo-opcodes by the Pascal Assembler.

E.1 Code File Organization

An assembly-language code file consists of a segment dictionary and a code part, as shown in Figure E-1:

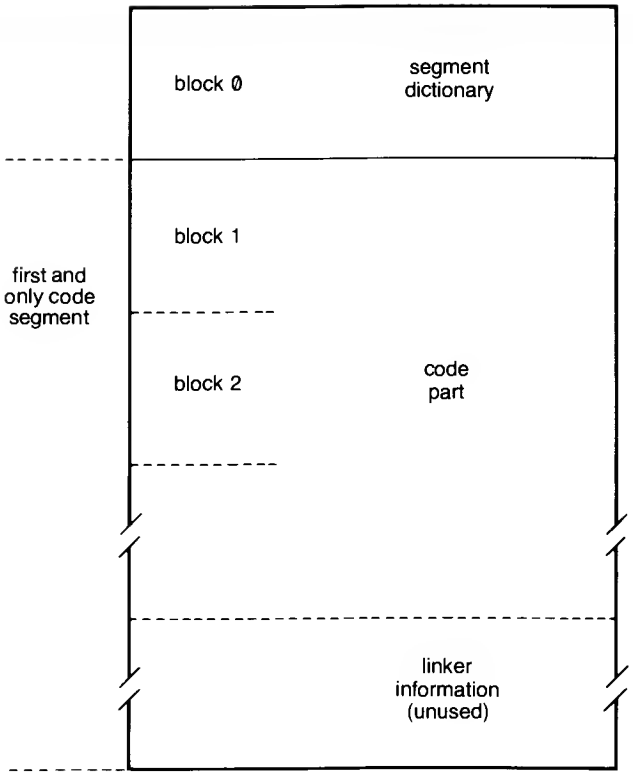


Figure E-1. An Assembly-Language Code File

The first block of a code file generated by the Pascal Assembler is in the standard format for block Ø of a Pascal code file; this block is called the *segment dictionary*. The remaining blocks of the file constitute the code part of the code file, which is a single code segment in this kind of file. The code part is followed by linker information: in an assembly-language code file, this information is unused.

Be especially careful in reading this section: words (two bytes of data) are used as well as bytes. Be sure you know which type each number refers to.

E.2 The Segment Dictionary

Since the code part is a single segment, most of the information in the segment dictionary is unused. Figure E-2 shows the information that is used.

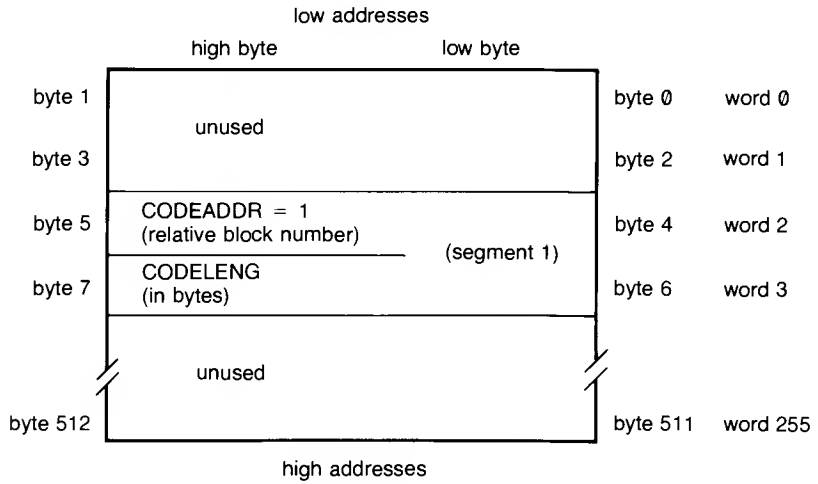


Figure E-2. A Segment Dictionary

Two 2-byte fields in this block are relevant when you write a module loader. The first starts at byte 4 and is the starting block number (relative to the beginning of the file) of the code generated by the Pascal Assembler; call this CODEADDR, because that is the field name in the Pascal declaration. The second starts at byte 6 and is the length, in bytes, of the code; call it CODELENG, for the same reason.

Your loading routine should begin loading at the relative block number (usually 1) indicated by CODEADDR, and should load the number of bytes indicated by CODELENG.

E.3 The Code Part of a Code File

Following the segment dictionary is the code part, which contains the procedure dictionary and the procedures themselves. This is diagrammed in Figure E-3.

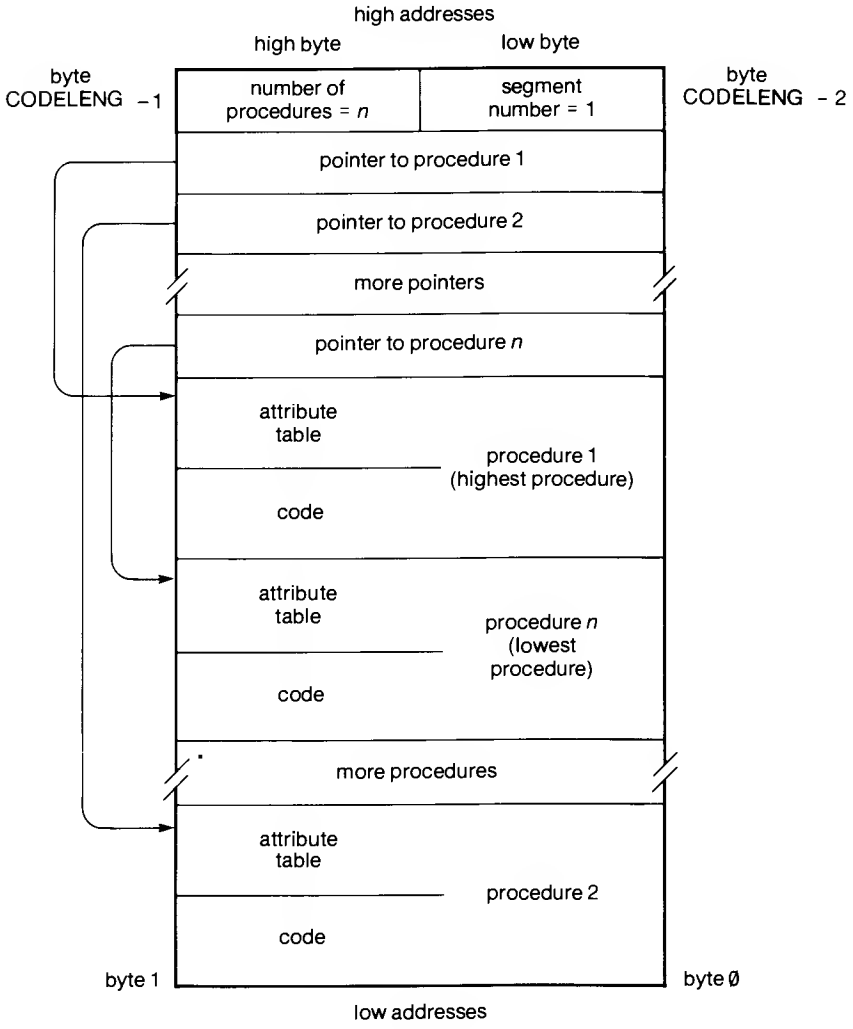


Figure E-3. The Code Part of a Code File

E.3.1 The Procedure Dictionary

The low byte of the last word of the precedure dictionary is at the address CODELENG-2; the structure grows down toward lower addresses, as shown in Figure E-3. To decipher the structure, look at the word whose location is calculated by CODEADDR * 512 + CODELENG-2. The low byte should contain 1. The high byte tells you the number of procedures in the code file. Each use of the psuedo-opcodes .PROC or .FUNC increments this number. Below this word is a sequence of words that contain self-relative pointers to the last word of each procedure in the code file.

A self-relative pointer contains the absolute distance, in bytes, between the low byte of the pointer and the low byte of the word to which it points. To find the address referred to by a self-relative pointer, subtract the value of the pointer from the address of its location.

The number of a procedure is an index into the procedure dictionary: the *n*th word in the dictionary (counting down from higher addresses) contains a pointer to the top (high address) of the code of procedure number *n*. As Ø is not a valid procedure number, the Øth word of the dictionary is used to store a Pascal-specific descriptor (usually 1) and the number of procedures in the code file (as described above).

E.3.2 Procedures

Each procedure consists of two parts: the procedure code, and the procedure attribute table. The procedure code is contained in the lower portion of the procedure and grows upward toward the higher addresses.

E.3.3 Assembly-Language Procedure Attribute Tables

A precedure's attribute table provides information needed to execute the procedure. Procedure attribute tables are pointed to by entries in the procedure dictionary of each code file.

The format of the attribute table of an assembly-language procedure is illustrated in Figure E-4.

The other type of attribute table is described in the *Apple III Pascal Technical Reference Manual*.

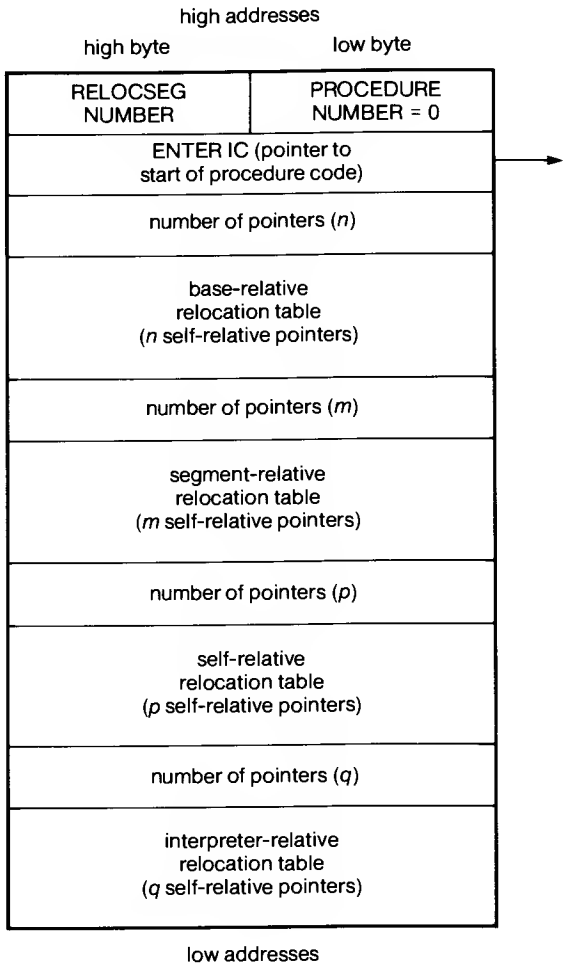


Figure E-4. An Assembly-Language Procedure Attribute Table

The highest word in the attribute table of an assembly-language procedure always has Ø in its PROCEDURE NUMBER field. This Ø can be used as a flag to indicate to your loading routine that relocation references may need changing to agree with the other information in the attribute table. The RELOCSEG NUMBER field must contain Ø.

The second-highest word of the attribute table is the ENTER IC field: a self-relative pointer to the first executable instruction of the procedure. Following this are four relocation tables; from high address to low address, they are base-relative, segment-relative, procedure-relative, and interpreter-relative.

E.3.4 Relocation Tables

A relocation table is a sequence of records that contain information necessary to relocate any relocatable addresses used by code within the procedure. These addresses must be relocated whenever the code file containing the procedure is loaded into or moved within memory.

The format of all four relocation tables is the same: the highest word of each table specifies the number of entries (possibly Ø) that follow at the lower addresses in the table. The remainder of each table contains the one-word self-relative pointers to locations in the procedure code that must be changed by the addition of the appropriate relative relocation constant, which is known to your interpreter when the code is loaded.

E.3.4.1 Base-Relative Relocation Table

Every reference to a label associated with the psuedo-opcodes .PUBLIC and .PRIVATE generates an entry into this table. In the Pascal environment, these opcodes flag references to data global to the Pascal program.

E.3.4.2 Segment-Relative Relocation Table

References to labels associated with .REF generate segment-relative relocation entries. The offsets in this table are relative to the beginning of the code portion of the code file: the address of the lowest byte of the code module is added to each of the addresses pointed to in the relocation table. Additionally, references to .PROC or .FUNC names generate entries into this table.

E.3.4.3 Procedure-Relative Relocation Table

Addresses pointed to by the procedure-relative relocation table must be relocated relative to the lowest address of the procedure. The address of the lowest byte in the procedure must be added to the contents of the words pointed to in the relocation table. The relevant Assembler directives are .BYTE, .WORD, .BLOCK, and .ASCII. Additionally, any non-relative reference (that is, JMP or LDA, but not BNE or BCS) generates an entry into this table.

E.3.4.4 Interpreter-Relative Relocation Table

Entries into this table are generated by references to labels defined by the .INTERP psuedo-opcode. The Pascal System uses this to index into a jump table in the interpreter.

Bibliography

These manuals, in addition to the present one, explain the workings of the Apple III and its system software:

Apple Business BASIC Reference Manual (Volumes 1 and 2). Cupertino, Calif.: Apple Computer, 1981.

Apple III Owner's Guide. Cupertino, Calif.: Apple Computer, 1982.

Apple III Pascal: Introduction, Filer, and Editor. Cupertino, Calif.: Apple Computer, 1981.

Apple III Pascal: Program Preparation Tools. Cupertino, Calif.: Apple Computer, 1981.

Apple III Pascal Programmer's Manual (Volumes 1 and 2). Cupertino, Calif.: Apple Computer, 1981.

Apple III SOS Device Driver Writer's Guide. Cupertino, Calif.: Apple Computer,Inc., 1982.

Apple III Standard Device Drivers Manual. Cupertino, Calif.: Apple Computer, 1981.

These books explain 65Ø2 assembly-language programming:

Leventhal, Lance A. *65Ø2 Assembly Language Programming*. Berkeley: Osborne/McGraw-Hill, 1979.

Scanlon, Leo J. *65Ø2 Software Design*. Indianapolis: Howard W. Sams & Co., 198Ø.

Zaks, Rodnay. *Programming the 65Ø2*. Berkeley: Sybex, 1978.

Index

Page references in Volume 2 are shown in square brackets [ ].

B

C

D

E

F

G

H

I

J

K

L

M

N

S

T

U

V

W

X

Y

Z

Special Symbols and Numbers